Светлин Наков и колектив

ПРОГРАМИРАНЕ ЗА .NET Framework

2 ТОМ

Българска асоциация на разработчиците на софтуер
telerik
НАПС Национална академия по разработка на софтуер

Кратко съдържание

Том 2

Програмиране за .NET Framework

Светлин Наков и колектив

Александър Русев
Александър Хаджикръстев
Антон Андреев
Бранимир Ангелов
Васил Бакалов
Виктор Живков
Галин Илиев
Георги Пенчев
Деян Варчев
Димитър Бонев
Димитър Канев
Ивайло Димов
Ивайло Христов
Иван Митев
Лазар Кирчев
Манол Донев
Мартин Кулов
Михаил Стойнов
Моника Алексиева
Николай Недялков
Панайот Добриков
Преслав Наков
Радослав Иванов
Рослан Борисов
Светлин Наков
Стефан Добрев
Стефан Захариев
Стефан Кирязов
Стоян Дамов
Тодор Колев
Христо Дешев
Христо Радков
Цветелин Андреев
Явор Ташев

Българска асоциация на разработчиците на софтуер

София, 2004-2006

Програмиране за .NET Framework (том 2)

Официален уеб сайт:

www.devbg.org/dotnetbook/

ISBN: 954-775-672-9

ISBN: 978-954-775-672-4

НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

Лекторите

» Светлин Наков е автор на десетки технически публикации и няколко книги, свързани с разработката на софтуер, заради което е търсен лектор и консултант.

Той е разработчик с дългогодишен опит, работил по разнообразни проекти, реализирани с различни технологии (.NET, Java, Oracle, PKI и др.) и преподавател по съвременни софтуерни технологии в СУ "Св. Климент Охридски".

През 2004 г. е носител на наградата "**Джон Атанасов**" на президента на България Георги Първанов.

Светлин Наков ръководи обучението по Java технологии в Академията.

» Мартин Кулов е софтуерен инженер и консултант с дългогодишен опит в изграждането на решения с платформите на Microsoft.

Мартин е опитен инструктор и сертифициран от Майкрософт разработчик по програмите MCSD, MCSD.NET, MCPD и MVP и международен лектор в световната организация на .NET потребителските групи INETA.

Мартин Кулов ръководи обучението по .NET технологии в Академията.

Академията

» **Национална академия по разработка на софтуер** (**НАРС**) е център за професионално обучение на софтуерни специалисти.

» **НАРС** провежда **БЕЗПЛАТНО** курсове по разработка на софтуер и съвременни софтуерни технологии в София и други градове.

» Предлагани специалности:

- **Въведение в програмирането (с езиците C# и Java)**
- **Core .NET Developer**
- **Core Java Developer**

» **Качествено обучение** с много **практически проекти** и индивидуално внимание за всеки.

» **Гарантирана работа!** Трудов договор при постъпване в Академията.

» БЕЗПЛАТНО!

Учите безплатно във въведителните курсове и по стипендии от работодателите в следващите нива.

http://academy.devbg.org

www.devbg.org

Българска асоциация на разработчиците на софтуер (БАРС) е нестопанска организация, която подпомага професионалното развитие на българските софтуерни специалисти чрез образователни и други инициативи.

БАРС работи за насърчаване обмяната на опит между разработчиците и за усъвършенстване на техните знания и умения в областта на проектирането и разработката на софтуер.

Асоциацията организира специализирани конференции, семинари и курсове за обучение по разработка на софтуер и софтуерни технологии.

БАРС организира създаването на Национална академия по разработка на софтуер – учебен център за професионална подготовка на софтуерни специалисти.

Отзив от Теодор Милев

Свидетели сме как платформата Microsoft .NET се налага все повече в света на софтуерните технологии. Тази тенденция се наблюдава и в България, където прогресивно нараства броят на проектите, реализирани на базата на .NET. С увеличаване на .NET разработчиците расте и нуждата от качествена техническа литература и учебни материали, които да бъдат използвани при обучението на .NET специалисти.

"Програмиране за .NET Framework" е първата чисто българска книга за Microsoft .NET технологиите. Тя представя на читателя в последователен, структуриран, достъпен и разбираем вид основните концепции за разработка на приложения с .NET Framework и езика C#. Книгата обхваща в детайли всички основни .NET технологии като набляга върху най-важните от тях – ADO.NET, ASP.NET, Windows Forms и XML уеб услуги.

По качество на изложения материал книгата се отличава с високо професионално ниво и превъзхожда повечето преводни издания по темата. Тя е отлично структурирана, а стилът на изложението е лесен за възприемане. Информацията е поднесена с много примери, а това е най-важното за един софтуерен разработчик.

Книгата е написана от широк екип доказани специалисти, работещи в партньорските фирми на Майкрософт – хора с опит в разработката на .NET приложения. Основният автор и ръководител на проекта, Светлин Наков, е изтъкнат .NET специалист, лектор в множество семинари и конференции, търсен консултант и преподавател. Негови са заслугите за курсовете по програмиране за платформа .NET във Факултета по математика и информатика на Софийски университет. Негови са и основните заслуги за целия проект по изготвяне на изчерпателно учебно съдържание и книга по програмиране за .NET Framework.

Светлин Наков е носител на най-голямото отличие в областта на информационните технологии – наградата "Джон Атанасов" на Президента Георги Първанов за принос към развитието на информационните технологии информационното общество. Той е автор на десетки статии и книги за програмиране, а настоящото издание е поредната му добра изява.

Настоящата книга е отлично учебно пособие както за начинаещи, така и за напреднали читатели, които имат желание и амбиции да станат професионални .NET разработчици.

Теодор Милев,
Управляващ директор на "Майкрософт България"

Отзив от Божидар Сендов

Книгата е оригинално българско творение, с нищо неотстъпващо по качество и обем на световните бестселъри с компютърна тематика. Материалът е поднесен достъпно и е богато илюстриран с примери, което я прави не само отлично въведение в платформата .NET за начинаещия, но и отличен справочник за професионалиста-програмист на C#. Читателят може да се запознае в детайли не само с общите принципи, но и с редица тънкости на програмирането за .NET. Широко застъпени са редица "универсални" теми като обектно-ориентирано програмиране, регулярни изрази, XML, релационни бази данни, програмиране в Интернет, многозадачност, сигурност и др.

Книгата се отличава със стегнат и ясен стил на изложението, като е постигнато завидно педагогическо майсторство. Това не бива да ни изненадва – авторите са водещи специалисти с богат опит не само като професионални софтуерни разработчици, но и като преподаватели във Факултета по математика и информатика (ФМИ) на СУ "Св. Климент Охридски". Самата книга в значителна степен се основава на работни лекции, използвани и проверени в поредица от курсове по програмиране за .NET Framework във ФМИ. Сайтът на книгата съдържа над 2000 безплатни слайда, следващи стриктно съдържанието ѝ, а книгата е напълно безплатна в електронния си вариант, което максимално улеснява използването ѝ в съответен курс по програмиране.

Не на последно място, заслужава да се отбележи систематичният опит за превод на всички термини на български език, съобразен с вече наложилата се българска терминология, но и с оригинални идеи при новите понятия.

Работата, която авторите са свършили, е наистина чудесна, а книгата е задължителна част от библиотеката на всеки с интерес към езика C# и изобщо към водещата платформа на Майкрософт .NET.

доц. д-р Божидар Сендов
Факултет по математика и Информатика,
Софийски Университет "Св. Климент Охридски"

Отзив от Стоян Йорданов

"Програмиране за .NET Framework" е уникално ръководство за платформата .NET. Въпреки, че не е учебник по програмиране, книгата е изключително подходяща както за начинаещия програмист, сблъскващ се за пръв път с .NET, така и за опитния разработчик на .NET приложения, целящ да систематизира и попълни знанията си. Всяка тема в "Програмиране за .NET Framework" започва с основите на разглежданите в нея технологии, но към края на темата читателят е вече запознат с детайлите и тънкостите, необходими за успешното им прилагане в практиката.

Обхващайки най-важните аспекти на .NET Framework, книгата започва от основите на езика C# и .NET платформата и постепенно достига до сложни концепции като уеб услуги, сигурност, сериализация, работа с отдалечени обекти, манипулиране на бази данни чрез ADO.NET, потребителски интерфейс с Windows Forms, ASP.NET уеб приложения и т.н. Информацията е поднесена изключително достъпно и подкрепена с многобройни примери и илюстрации. Всяка тема включва и упражнения за самостоятелна работа – неотменим елемент за затвърдяване на придобитите от нея знания.

Авторският колектив включва утвърдени специалисти от софтуерните среди. Въпреки, че авторите са над 30, "Програмиране за .NET Framework" не е просто сборник от статии; напротив – всеки от тях е допринесъл с опита и труда си, за да може книгата да бъде това, което е – добре структурирано и изчерпателно ръководство.

Учебник за студента или справочник за специалиста – "Програмиране за .NET Framework" е задължителна за библиотеката на всеки който има досег с .NET.

Стоян Йорданов,
Software Design Engineer,
Microsoft Corpartion (Redmond)

* Мнението е лично на автора му и не обвързва Microsoft Corporation по никакъв начин

НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

Лекторите

» Светлин Наков е автор на десетки технически публикации и няколко книги, свързани с разработката на софтуер, заради което е търсен лектор и консултант.

Той е разработчик с дългогодишен опит, работил по разнообразни проекти, реализирани с различни технологии (.NET, Java, Oracle, PKI и др.) и преподавател по съвременни софтуерни технологии в СУ "Св. Климент Охридски".

През 2004 г. е носител на наградата "**Джон Атанасов**" на президента на България Георги Първанов.

Светлин Наков ръководи обучението по Java технологии в Академията.

» Мартин Кулов е софтуерен инженер и консултант с дългогодишен опит в изграждането на решения с платформите на Microsoft.

Мартин е опитен инструктор и сертифициран от Майкрософт разработчик по програмите MCSD, MCSD.NET, MCPD и MVP и международен лектор в световната организация на .NET потребителските групи INETA.

Мартин Кулов ръководи обучението по .NET технологии в Академията.

Академията

» **Национална академия по разработка на софтуер** (**НАРС**) е център за професионално обучение на софтуерни специалисти.

» **НАРС** провежда **БЕЗПЛАТНО** курсове по разработка на софтуер и съвременни софтуерни технологии в София и други градове.

» Предлагани специалности:

- **Въведение в програмирането (с езиците C# и Java)**
- **Core .NET Developer**
- **Core Java Developer**

» **Качествено обучение** с много **практически проекти** и индивидуално внимание за всеки.

» **Гарантирана работа!** Трудов договор при постъпване в Академията.

» БЕЗПЛАТНО!

Учите безплатно във въведителните курсове и по стипендии от работодателите в следващите нива.

http://academy.devbg.org

Съдържание

Том 2

НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

Лекторите

» ***Светлин Наков*** е автор на десетки технически публикации и няколко книги, свързани с разработката на софтуер, заради което е търсен лектор и консултант.

Той е разработчик с дългогодишен опит, работил по разнообразни проекти, реализирани с различни технологии (.NET, Java, Oracle, PKI и др.) и преподавател по съвременни софтуерни технологии в СУ "Св. Климент Охридски".

През 2004 г. е носител на наградата "**Джон Атанасов**" на президента на България Георги Първанов.

Светлин Наков ръководи обучението по Java технологии в Академията.

» ***Мартин Кулов*** е софтуерен инженер и консултант с дългогодишен опит в изграждането на решения с платформите на Microsoft.

Мартин е опитен инструктор и сертифициран от Майкрософт разработчик по програмите MCSD, MCSD.NET, MCPD и MVP и международен лектор в световната организация на .NET потребителските групи INETA.

Мартин Кулов ръководи обучението по .NET технологии в Академията.

Академията

» **Национална академия по разработка на софтуер** (**НАРС**) е център за професионално обучение на софтуерни специалисти.

» **НАРС** провежда **БЕЗПЛАТНО** курсове по разработка на софтуер и съвременни софтуерни технологии в София и други градове.

» Предлагани специалности:

- **Въведение в програмирането (с езиците C# и Java)**
- **Core .NET Developer**
- **Core Java Developer**

» **Качествено обучение** с много **практически проекти** и индивидуално внимание за всеки.

» **Гарантирана работа!** Трудов договор при постъпване в Академията.

» БЕЗПЛАТНО!

Учите безплатно във въведителните курсове и по стипендии от работодателите в следващите нива.

http://academy.devbg.org

www.devbg.org

Българска асоциация на разработчиците на софтуер (БАРС) е нестопанска организация, която подпомага професионалното развитие на българските софтуерни специалисти чрез образователни и други инициативи.

БАРС работи за насърчаване обмяната на опит между разработчиците и за усъвършенстване на техните знания и умения в областта на проектирането и разработката на софтуер.

Асоциацията организира специализирани конференции, семинари и курсове за обучение по разработка на софтуер и софтуерни технологии.

БАРС организира създаването на Национална академия по разработка на софтуер – учебен център за професионална подготовка на софтуерни специалисти.

Предговор към втория том

Ако по принцип не четете уводите на книгите, пропуснете и този. В него ще научите най-вече какво ви предстои в следващите глави и как се стигна до написването на настоящата книга.

Това е втори том на първата чисто българска книга за програмиране с .NET Framework и C#, но въпреки, че фокусира върху .NET Framework 1.1, тя е едно от най-полезните четива в тази област. Написана от специалисти с опит както в практическата работа с .NET, така и в обучението по програмиране, книгата ще ви даде не само основите на .NET програмирането, но и ще ви запознае с някои по-сложни концепции и ще ви предаде от опита на авторите.

За кого е предназначена тази книга?

Вторият том на книгата е предназначен за всички, които са прочели първия том и той им допада. Тя е за всички, които искат да продължат обогатяването на знанията и уменията си за разработка на софтуер за .NET платформата.

Вторият том е просто продължение на първия и включва няколко много важни технологии от .NET Framework, а именно Windows Forms, ASP.NET уеб приложения и уеб услуги.

Тази книга ще ви даде много повече от начални знания. Тя ще ви предаде опит, натрупан в продължение години, и ще ви запознае с утвърдените практики при използването на .NET технологиите.

Книгата е полезна не само за .NET програмисти, но и за всички, които имат желание да се занимават сериозно с разработка на софтуер. В нея се обръща внимание не само на специфичните .NET технологии, но и на някои фундаментални концепции, които всеки програмист трябва добре да знае и разбира.

Необходими начални познания

Тази книга не е подходяща за хора, които никога не са програмирали в живота си. Ако сте абсолютно начинаещ, спрете да четете и просто започнете с друга книга!

Том 2 на книгата не е подходящ за хора, които не са чели (или поне прегледали набързо) първия том. Вторият том е естествено продължение

на първия том и е силно свързан с материала, изложен в него. И двете части на книгата са свободно достъпни от Интернет (от адрес http://www.devbg.org/dotnetbook/), така че нямате оправдание да започвате направо от втората. Не ви го препоръчваме!

Какво обхваща вторият том на тази книга?

Програмирането за .NET Framework изисква познания на неговите базови концепции (модел на изпълнение на кода, обща система от типове, управление на паметта, масиви, колекции, символни низове и др.), както и познаване на често използваните технологии – ADO.NET (за достъп до бази от данни), Windows Forms (за приложения с графичен потребителски интерфейс), ASP.NET (за уеб приложения и уеб услуги) и др.

Първият том на книгата обхваща основните концепции в .NET програмирането (от езика C# до ADO.NET), а вторият – по-сложните технологии като Windows Forms, ASP.NET, уеб услуги, нишки, мрежово програмиране, сигурност и др.

Във втория том се обръща внимание на създаването на графичен потребителски интерфейс с Windows Forms и уеб-базирани приложения с ASP.NET. Ще бъдат разгледани и някои по-сложни концепции като отражение на типовете, сериализация, многонишково програмиране, уеб услуги, отдалечено извикване на методи (remoting), взаимодействие с неуправляван код, асемблита, управление на сигурността, по-важни инструменти за разработка и др. Ще бъде разгледана и свободната имплементация на .NET Framework за Linux и други операционни системи Mono. Накрая ще бъде описана разработката на един цялостен практически проект, който обхваща всички по-важни технологии и демонстрира добрите практики при изграждането на .NET приложения.

Фокусът е върху .NET Framework 1.1

Всички теми са базирани на .NET Framework 1.1, Visual Studio .NET 2003 и MS SQL Server 2000. За съжаление по време на изготвянето на текста на книгата (през 2004-2005 г.) версия 2.0 на .NET платформата едва прохождаше и това наложи да не бъдат включени новостите от него.

Надяваме се в следващото издание на книгата авторският колектив да намери време и сили да обнови съдържанието с новостите от .NET 2.0 и да отправи поглед към .NET 3.0.

Как е представена информацията?

Въпреки големия брой автори, съавтори и редактори, стилът на текста в книгата е изключително достъпен. Съдържанието е представено в добре структуриран вид, разделено с множество заглавия и подзаглавия, което позволява лесното му възприемане, както и бързото търсене на информация в текста.

Настоящата книга е написана от програмисти за програмисти. Авторите са действащи софтуерни разработчици, хора с реален опит както в разработването на софтуер, така и в обучението по програмиране. Благодарение на това качеството на изложението е на много високо ниво.

Всички автори ясно съзнават, че примерният сорс код е едно от най-важните неща в една книга за програмиране. Именно поради тази причина текстът е съпроводен с много, много примери, илюстрации и картинки.

Въобще някой чете ли текста, когато има добър и ясен пример? Повечето програмисти първо гледат дали примерът ще им свърши работа, и само ако нещо не е ясно, се зачитат в текста (това всъщност не е никак добра практика, но такава е реалността). Ето защо многото и добре подбрани примери са един от най-важните принципи, залегнали в тази книга.

Поглед към съдържанието на втория том

Книгата се състои от 29 глави, които поради големия обем са разделени в два тома. Том 1 съдържа първите 14 глави, а том 2 – останалите 15. Това важи само за хартиеното издание на книгата. В електронния вариант тя се разпространява като едно цяло.

Нека направим кратък преглед на всяка една от главите и да се запознаем с нейното съдържание, за да разберем какво ни очаква по-нататък. Главите от втория том можете да намерите в настоящото издание, а останалите – в първи том.

Глава 15. Графичен потребителски интерфейс с Windows Forms

В глава 15 се разглеждат средствата на Windows Forms за създаване на прозоречно-базиран графичен потребителски интерфейс (GUI) за .NET приложенията. Представят се програмният модел на Windows Forms, неговите базови контроли, средствата за създаване на прозорци, диалози, менюта, ленти с инструменти и статус ленти, както и някои по-сложни концепции като: MDI приложения, data-binding, наследяване на форми, хостинг на контроли в Internet Explorer, работа с нишки във Windows Forms и др.

Автори на главата са Радослав Иванов (по-голямата част) и Светлин Наков. Текстът е базиран на лекцията на Светлин Наков по същата тема. Редактори са Светлин Наков и Пламен Табаков.

Глава 16. Изграждане на уеб приложения с ASP.NET

В глава 16 се разглежда разработката на уеб приложения с ASP.NET. Представят се програмният модел на ASP.NET, уеб формите, кодът зад тях, жизненият цикъл на уеб приложенията, различните типове контроли и техните събития. Показва се как се дебъгват и проследяват уеб прило-

жения. Отделя се внимание на валидацията на данни, въведени от потребителя. Разглежда се концепцията за управление на състоянието на обектите – View State и Session State. Демонстрира се как могат да се визуализират и редактират данни, съхранявани в база от данни. Дискутират се разгръщането и конфигурирането на ASP.NET уеб приложенията в Internet Information Server (IIS) и сигурността при уеб приложенията.

Автори на главата са Михаил Стойнов, Рослан Борисов, Стефан Добрев, Деян Варчев, Иван Митев и Христо Дешев. Текстът е базиран на лекцията на Михаил Стойнов по същата тема. Редактори са Иван Митев и Пламен Табаков.

Тази глава беше най-обемната, най-трудната и най-бавно написаната. Поради някои проблемни ситуации в авторския колектив се наложи на няколко пъти да се сменят авторите и това реално забави целия втори том. За радост всичко приключи успешно.

Глава 17. Многонишково програмиране и синхронизация

В глава 17 се разглежда многозадачността в съвременните операционни системи и средствата за паралелно изпълнение на програмен код, които .NET Framework предоставя. Обръща се внимание на нишките (threads), техните състояния и управлението на техния жизнен цикъл – стартиране, приспиване, събуждане, прекратяване и др.

Разглеждат средствата за синхронизация на нишки при достъп до общи данни, както и начините за изчакване на зает ресурс и нотификация при освобождаване на ресурс. Обръща се внимание както на синхронизационните обекти в .NET Framework, така и на неуправляваните синхронизационни обекти от операционната система.

Изяснява се концепцията за работа с вградения в .NET Framework пул от нишки (thread pool), начините за асинхронно изпълнение на задачи, средствата за контрол над тяхното поведение и препоръчваните практики за работа с тях.

Автор на главата е Александър Русев. Текстът е базиран в голямата си част на лекцията на Михаил Стойнов и авторските бележки в нея. редактори са Иван Митев, Георги Митев, Георги Митев, Яни Георгиев и Минчо Колев.

Глава 18. Мрежово и Интернет програмиране

В глава 18 се разглеждат някои основни средства, предлагани от .NET Framework за мрежово програмиране. Главата започва със съвсем кратко въведение в принципите на работа на съвременните компютърни мрежи и на Интернет и продължава с протоколите, чрез които се осъществява мрежовата комуникация. Обект на дискусия са както класовете за работа с TCP и UDP сокети, така и някои класове, предлагащи по-специфични възможности, като представяне на IP адреси, изпълняване на DNS заявки и

др. В края на главата ще се представят средствата за извличане на уеб-ресурси от Интернет и на класовете за работа с e-mail в .NET Framework.

Автори на главата са Ивайло Христов и Георги Пенчев. Текстът широко използва лекцията на Ивайло Христов по същата тема. Редактори са Венцислав Попов, Стефан Чанков, Лъчезар Георгиев и Теодор Стоев.

Глава 19. Отражение на типовете (Reflection)

В глава 19 се представя понятието Global Assembly Cache (GAC) и отражение на типовете (reflection). Разглеждат се начините за зареждане на асемблии. Демонстрира се как може да се извлече информация за типовете в дадено асемблии и за членовете на даден тип. Разглеждат се начини за динамично извикване на членове от даден тип. Обяснява се как може да се създаде едно асемблии, да се дефинират типове в него и асемблито да се запише във файл по време на изпълнение на програмата.

Автор на главата е Димитър Канев. Текстът е базиран на лекцията на Ивайло Христов по същата тема. Редактор е Светлин Наков.

Глава 20. Сериализация на данни

В глава 20 се разглежда сериализацията на данни в .NET Framework. Обяснява се какво е сериализация, за какво се използва и как се контролира процесът на сериализация. Разглеждат се видовете форматери (formatters). Обяснява се какво е XML сериализация, как работи тя и как може да се контролира изходният XML при нейното използване.

Автор на главата е Радослав Иванов. Текстът е базиран на лекцията на Михаил Стойнов по същата тема. Редактор е Светлин Наков.

Глава 21. Уеб услуги с ASP.NET

В глава 21 се разглеждат уеб услугите, тяхното изграждане и консумация чрез ASP.NET и .NET Framework. Обект на дискусия са основните технологии, свързани с уеб услугите, и причината те да се превърнат в стандарт за интеграция и междуплатформена комуникация. Представят се различни сценарии за използването им. Разглежда се програмният модел за уеб услуги в ASP.NET и средствата за тяхното изграждане, изпълнение и разгръщане (deployment). Накрая се дискутират някои често срещани проблеми и утвърдени практики при разработката на уеб услуги чрез .NET Framework.

Автори на главата са Стефан Добрев и Деян Варчев. В текста са използвани материали от лекцията на Светлин Наков по същата тема. Технически редактор е Мартин Кулов.

Глава 22. Отдалечено извикване на методи (Remoting)

В глава 22 се разглежда инфраструктурата за отдалечени извиквания, която .NET Framework предоставя на разработчиците. Обясняват се основите на Remoting технологията и всеки един от нейните компоненти: канали, форматери, отдалечени обекти и активация. Дискутират се разликите между различните типове отдалечени обекти. Обясняват се техният жизнен цикъл и видовете маршализация. Стъпка по стъпка се достига до създаването на примерен Remoting сървър и клиент. Накрая се представя един гъвкав и практичен начин за конфигуриране на цялата Remoting инфраструктура чрез конфигурационни файлове.

Автор на главата е Виктор Живков. В текста са използвани материали от лекцията на Светлин Наков. Редактори са Иван Митев и Светлин Наков.

Глава 23. Взаимодействие с неуправляван код

Глава 23 разглежда как можем да разширим възможностите на .NET Framework чрез употреба на предоставените от Windows приложни програмни интерфейси (API). Дискутират се средствата за извикване на функционалност от динамични Win32 библиотеки и на проблемите с преобразуването (маршализацията) между Win32 и .NET типовете.

Обръща се внимание на връзката между .NET Framework и COM (компонентният модел на Windows). Разглеждат се както извикването на COM обекти от .NET код, така и разкриването на .NET компонент като COM обект. Демонстрира се и технологията IJW за използване на неуправляван код от програми, написани на Managed C++.

Автор на главата е Мартин Кулов. Текстът е базиран на неговата лекция по същата тема. Технически редактор е Галин Илиев.

Глава 24. Управление на паметта и ресурсите

В глава 24 се разглежда писането на правилен и ефективен код по отношение използването на паметта и ресурсите в .NET Framework. В началото се прави сравнение на предимствата и недостатъците на ръчното и автоматичното управление на памет и ресурси. След това се разглежда подробно автоматичното им управление с фокус най-вече върху системата за почистване на паметта в .NET (т. нар. garbage collector). Обръща се внимание на взаимодействието с нея и практиките, с които можем да ѝ помогнем да работи възможно най-ефективно.

Автори на главата са Стоян Дамов и Димитър Бонев. Технически редактор е Светлин Наков.

Глава 25. Асемблита и разпространение (deployment)

В глава 25 се разглежда най-малката съставна част на .NET приложенията – асембли, различните техники за разпространение на готовия софтуерен продукт на клиентските работни станции и някои избрани техники за създаване на инсталационни пакети и капаните, за които трябва да се внимава при създаване на инсталационни пакети.

Автор на тази глава е Галин Илиев. В текста е използвана частично лекцията на Михаил Стойнов. Редактор е Явор Янев.

Глава 26. Сигурност в .NET Framework

В глава 26 се разглежда как .NET Framework подпомага сигурността на създаваните приложения. Това включва както безопасност на типовете и защита на паметта, така и средствата за защита от изпълнение на нежелан код, автентикация и оторизация, електронен подпис и криптография. Разглеждат се технологиите на .NET Framework като Code Access Security, Role-Based Security, силно-именувани асемблита, цифрово подписване на XML документи (XMLDSIG) и други.

Автори на главата са Тодор Колев и Васил Бакалов. В текста е широко използвана лекцията на Светлин Наков по същата тема. Технически редактор е Станислав Златинов.

Глава 27. Mono - свободна имплементация на .NET

В глава 27 се разглежда една от алтернативите на Microsoft .NET Framework – проектът с отворен код Mono. Обясняват се накратко начините за инсталиране и работа с Mono, използването на вградените технологии ASP.NET и ADO.NET, както и създаването на графични приложения. Дават се и няколко съвети и препоръки за писането на преносим код.

Автори на главата са Цветелин Андреев и Антон Андреев. Текстът е базиран на лекцията на Антон Андреев по същата тема. Технически редактор е Светлин Наков. Като редактори участват още Соня Бибиликова, Мартин Кирицов, Николай Митев и Александър Николов.

Глава 28. Помощни инструменти за .NET разработчици

В глава 28 се разглеждат редица инструменти, използвани при разработката на .NET приложения. С тяхна помощ може значително да се улесни изпълнението на някои често срещани програмистки задачи. Изброените инструменти помагат за повишаване качеството на кода, за увеличаване продуктивността на разработка и за избягване на някои традиционни трудности при поддръжката. Разглеждат се в детайли инструментите .NET Reflector, FxCop, CodeSmith, NUnit (заедно с допълненията към него NMock, NUnitAsp и NUnitForms), log4net, NHibernate и NAnt.

Автори на главата са Иван Митев и Христо Дешев. Текстът е по техни авторски материали. Редактори са Теодора Пулева и Борислав Нановски.

Глава 29. Практически проект

В глава 29 се дискутира как могат да се приложат на практика технологиите, разгледани в предходните теми. Поставена е задача да се разработи един сериозен практически проект – система за запознанства в Интернет с възможност за уеб и GUI достъп.

При реализацията на системата се преминава през всичките фази от разработката на софтуерни проекти: анализиране и дефиниране на изискванията, изготвяне на системна архитектура, проектиране на база от данни, имплементация, тестване и внедряване на системата.

При изготвяне на архитектурата приложението се разделя на три слоя – база от данни (която се реализира с MS SQL Server 2000), бизнес слой (който се реализира като ASP.NET уеб услуга) и клиентски слой (който се реализира от две приложения – ASP.NET уеб клиент и Windows Forms GUI клиент).

Ръководител на проекта е Ивайло Христов. Автори на проекта са: Ивайло Христов (отговорен за Windows Forms клиента), Тодор Колев и Ивайло Димов (отговорни за уеб услугата и базата данни) и Бранимир Ангелов (отговорен за ASP.NET уеб клиента). Инсталаторът на проекта е създаден от Галин Илиев. Технически редактори на кода са Мартин Кулов, Светлин Наков, Стефан Добрев и Деян Варчев.

Автори на текста са Ивайло Христов, Тодор Колев, Ивайло Димов и Бранимир Ангелов. Технически редактор е Иван Митев. Редактор на текста е Вера Моллова.

Авторският колектив

Авторският колектив се състои от над 30 души – автори, съавтори, редактори и други. Ще представим всеки от тях с по няколко изречения (подредбата е по азбучен ред).

Александър Русев

Александър Русев е програмист във фирма Johnson Controls (www.jci.com), където се занимава с разработка на софтуер за леки автомобили. Завършил е Технически университет – София, специалност компютърни системи и технологии. Александър се е занимавал и с разработка на софтуер за мобилни телефони. Професионалните му интереси включват Java технологиите и .NET платформата. Можете да се свържете с Александър по e-mail: arussev@gmail.com.

Александър Хаджикръстев

Александър Хаджикръстев е софтуерен архитект със сериозен опит в областта на проектирането и разработката на уеб базирани системи и e-commerce приложения. Той е сътрудник и консултант на PC Magazine България (www.sagabg.net/PCMagazine/) и почетен член на Българската асоциация на софтуерните разработчици (www.devbg.org). Александър има дългогодишен опит като ръководител на софтуерни проекти във фирми, базирани в България и САЩ. Професионалните му интереси са свързани с проектирането и изграждането на .NET приложения, разработването на експертни системи и софтуер за управление и автоматизация на бизнес процеси.

Антон Андреев

Антон Андреев работи като ASP.NET уеб разработчик във фирма Elements of Art (www.eoa.bg). Той се интересува се от всичко, свързано с компютрите и най-вече с .NET и Linux. Като ученик се е занимавал с алгоритми и е участвал в олимпиади по информатика. Завършил е математическа гимназия и езикова гимназия с английски език, а в момента е студент в специалност информатика във Факултета по математика и информатика (ФМИ) на Софийски университет "Св. Климент Охридски". Работил е и като системен администратор във ФМИ и сега продължава да подпомага проектите на факултета, разработвайки нови сайтове. Неговият личен сайт е достъпен от адрес: http://debian.fmi.uni-sofia.bg/~toncho/portfolio/. Можете да се свържете с Антон по e-mail: anton.andreev@fmi.uni-sofia.bg.

Бранимир Ангелов

Бранимир Ангелов е софтуерен разработчик във фирма Gugga (www.gugga.net) и студент във Факултета по Математика и информатика на Софийски университет "Св. Климент Охридски", специалност компютърни науки. Неговите професионални интереси са в областта на обектно-ориентирания анализ, моделиране и програмиране, уеб технологиите и в частност изграждането на RIA (Rich Internet Applications) и разработката на софтуер за мобилни устройства. Бранимир е печелил грамоти и отличия от различни състезания, както и първо място на Националната олимпиада по информационни технологии, на която е бил и жури година по-късно.

Васил Бакалов

Васил Бакалов е студент, последен курс, в Американския университет в България, специалност Информатика. Той е председател на студентския клуб по информационни технологии и е студент-консултант на Microsoft България за университета. В рамките на клуба се занимава с управление на проекти и консултации по изпълнението им. Като студент-консултант на Microsoft България Васил подпомага усилията на Microsoft да поддържа тясна връзка със студентите и да ги информира и обучава по най-новите й

продукти и технологии. Васил работи и като сътрудник на PC Magazine България от няколко години и има редица статии и коментари в изданието. В университета той предлага и изготвя план за курс по практическо изучаване на роботика, като разширение на обучението по изкуствен интелект, който е одобрен и внедрен. Той работи и с няколко ИТ фирми, където изгражда решения, базирани на .NET платформата. Притежава професионална сертификация от Microsoft. Можете да се свържете с Васил по e-mail: dotnetbook@vassil.info.

Виктор Живков

Виктор Живков е софтуерен инженер в Интерконсулт България (www.icb.bg). В момента е студент в Софийски Университет "Св. Климент Охридски", специалност информатика. Професионалните му интереси са основно в областта на решенията, базирани на софтуер от Microsoft. Виктор има сериозен опит в работата с .NET Framework, Visual Studio .NET и Microsoft SQL Server. Той участва в проекти за различни информационни системи, главно за Норвегия. Членува в БАРС от 2005 година. За връзка с Виктор можете да използвате неговия e-mail: viktor.zhivkov@gmail.com.

Деян Варчев

Деян Варчев е старши уеб разработчик във фирма Vizibility (www.vizibility.net). Неговите отговорности включват проектирането и разработката на уеб базирани приложения, използващи последните технологии на Microsoft, проучване на новопоявяващи се технологии и планиране на тяхното внедряване в производството, както и обучение на нови колеги. Неговите професионални интереси са свързани тясно с технологиите на Microsoft – .NET платформата, SQL Server, IIS, BizTalk и др. Деян е студент по информатика във Факултета по математика и информатика на Софийски университет "Св. Климент Охридски".

Димитър Бонев

Димитър Бонев е софтуерен разработчик във фирма Formula Telecom Solutions (www.fts-soft.com). Той отговаря за разработването на уеб базирани приложения за корпоративни клиенти, както и за някои модули и инструменти, свързани с вътрешния процес на разработка във фирмата. Професионалните му интереси са насочени предимно към .NET платформата, методологията extreme programming и софтуерния дизайн. Димитър е завършил ВВВУ "Г. Бенковски", специалност компютърна техника. Той има богат опит в разработването на софтуерни решения, предимно с технологиите на Microsoft и Borland.

Димитър Канев

Димитър Канев е разработчик на софтуер във фирма Медсофт (www.medsoft.biz). Той е завършил Факултета по математика и информатика на Софийски университет "Св. Климент Охридски", специалност

информатика. Професионалните му интереси са основно в областта на решенията, базирани на софтуер от Microsoft. Димитър има сериозен опит в работата с Visual Studio .NET, Microsoft SQL Server и ГИС системи. Работил е в проекти за изграждане на големи информационни системи, свързани с ГИС решения, и експертни системи за медицински лаборатории.

Галин Илиев

Галин Илиев е ръководител на проекти и софтуерен архитект в българския офис на Technology Services Consulting Group (www.wordassist.com). Галин е участвал в проектирането и разработването на големи информационни системи, Интернет сайтове с управление на съдържанието, допълнения и интеграция на MS Office със системи за управление на документи. Той притежава степен бакалавър по мениджмънт и информационни технологии, а също и сертификация MCSD за Visual Studio 6.0 и Visual Studio .NET. Той има сериозен опит с работата с Visual Studio .NET, MS SQL Server, MS IIS и MS Exchange. Личният му сайт е достъпен от адрес www.galcho.com, а e-mail адресът му е Iliev@galcho.com.

Георги Пенчев

Георги Пенчев е софтуерен разработчик във фирма Symex България (www.symex.bg), където отговаря за разработка на финансово ориентирани графични Java приложения и на Интернет финансови портали с Java и PHP. Участвал е в изграждането на продукти за следене и обработка на борсови индекси и котировки за Българската фондова борса. Георги е студент по информатика във Факултета по математика и информатика на Софийски университет "Св. Климент Охридски". Професионалните и академичните му интереси са насочени към Java и .NET технологиите, биоинформатиката, теоретичната информатика, изкуствения интелект и базите от знания. През 2004 и 2005 г. е асистент в курса по "Информационни технологии" за студенти с нарушено зрение и в практическия курс по "Структури от данни и програмиране" в Софийски университет. Можете да се свържете с Георги по e-mail: pench_wot@yahoo.com.

Иван Митев

Иван Митев е софтуерен разработчик във фирма EON Technologies (www.eontechnologies.bg). Той е завършил Факултета по математика и информатика на Софийски университет "Св. Климент Охридски", специалност информатика. Иван е участвал в проектирането и реализацията на множество информационни системи, основно ГИС решения. Професионалният му опит е в разработки предимно с продукти и технологии на Microsoft. Основните интереси на Иван са в създаването на качествени и ефективни софтуерни решения чрез използването на подходящи практики, технологии и инструменти. Технически уеблог, който той поддържа от началото на 2004 година, е с акцент върху .NET програмирането и е достъпен на адрес http://immitev.blogspot.com. Можете да се свържете с Иван по e-mail: immitev@gmail.com.

Ивайло Димов

Ивайло Димов е софтуерен разработчик във фирма Gugga (www.gugga.com). Неговите интереси са в областта на обектно-ориентираното моделиране, програмиране и анализ, базите от данни, уеб приложенията и приложения, базирани на Microsoft .NET Framework. В момента Ивайло е студент във Факултета по математика и информатика на Софийски университет "Св. Климент Охридски", специалност Компютърни науки. Той е сертифициран от Microsoft разработчик и е печелил редица грамоти и отличия от състезания по програмиране. През 2004 г. е победител в Националната олимпиада по информационни технологии и е участвал в журито на същата олимпиада година по-късно.

Ивайло Христов

Ивайло Христов е преподавател в Софийски университет "Св. Климент Охридски", където води курсове по "Програмиране за .NET Framework", "Качествен програмен код", "Увод в програмирането", "Обектно-ориентирано програмиране" и "Структури от данни в програмирането". Неговите професионални интереси са в областта на .NET технологиите и Интернет технологиите. Като ученик Ивайло е участник в редица национални състезания и конкурси по програмиране и е носител на престижни награди и отличия. Той участва в екип, реализирал образователен проект на Microsoft Research в областта на .NET Framework. Личният сайт на Ивайло е достъпен от адрес: www.ivaylo-hristov.net.

Лазар Кирчев

Лазар Кирчев е завършил Факултета по математика и информатика на Софийски университет "Св. Климент Охридски" и в момента е дипломант в специализация "Информационни системи". Той работи в Института за паралелна обработка на информацията към БАН по съвместен проект между Факултета по математика и информатика и БАН за изграждане на grid система. Неговите интереси включват .NET платформата, grid системите и базите от данни.

Манол Донев

Манол Донев е софтуерен разработчик във фирма **t**elerik (www.telerik.com). Той е част от екипа, който разработва уеб-базираната система за управление на съдържание Sitefinity (www.sitefinity.com). Манол е студент във Факултета по математика и информатика на Софийски университет "Св. Климент Охридски", специалност Информатика. Неговите професионални интереси обхващат най-вече .NET технологиите (в частност ASP.NET уеб приложения, XML и уеб услуги). Можете да се свържете с Манол по e-mail: manol.donev@gmail.com.

Мартин Кулов

Мартин Кулов е сертифициран инструктор и разработчик по програмите Microsoft Certified Trainer (MCT) и MCSD.NET. През 2006 г. е награден от Майкрософт с наградата Most Valuable Professional (MVP). Той е директор направление .NET към Национална академия по разработка на софтуер, където е отговорен за разработка на курсове, обучение и проучване на най-новите технологии на Майкрософт като Visual Studio Team System, Indigo, WSE, ASP.NET, Analysis Services 2005, VSTO, Atlas и др. Мартин е почетен член на Българската асоциация на разработчиците на софтуер (БАРС), член на SofiaDev .NET потребителската група, лектор при международната .NET асоциация - INETA и лектор на редица семинари на Майкрософт. Той е регионален президент на Международната асоциация на софтуерните архитекти (IASA) за България. Неговият личен дневник (блог) може да намерите на адрес http://www.codeattest.com/blogs/martin.

Михаил Стойнов

Михаил Стойнов е софтуерен разработчик във фирма MPS (www.mps.bg), която е подизпълнител на Siemens A.G. Той се занимава професионално с програмиране за платформите Java и .NET Framework от няколко години. Участва като лектор в преподавателския екип на курсовете "Програмиране за .NET Framework" и "Качествен програмен код". Той е студент-консултант на Майкрософт България за Софийски университет през последните 2 години и подпомага разпространението на най-новите продукти и технологии на Microsoft в университета. Михаил е бил лектор на международни конференции за ГИС системи. Интересите му обхващат разработка на уеб приложения, приложения с бази от данни, изграждане на сървърни системи и участие в академични дейности.

Моника Алексиева

Моника Алексиева е софтуерен разработчик във фирма Солвер / Мидакс (www.midax.com). В момента следва специалност информатика във Факултета по математика и информатика на Софийски университет "Св. Климент Охридски". Моника има професионален опит в разработката за .NET Framework с езика C# и е сертифициран от Microsoft разработчик за .NET платформата. Нейните интереси са в областта на технологиите за изграждането на графичен потребителски интерфейс и разработката на приложения за мобилни устройства. През 2004 година Моника е асистент по "Структури от Данни" в Софийски университет.

Николай Недялков

Николай Недялков е президент на Асоциацията за информационна сигурност (www.iseca.org) която е създадена с цел прилагане на най-добрите практики за осигуряване на информационната сигурност на национално ниво и при извършването на електронен бизнес. Николай е професионален разработчик на софтуер, консултант и преподавател с дългогодишен

опит. Той е автор на статии и лектор на множество конференции и семинари в областта на софтуерните технологии и информационна сигурност. Преподавателският му опит се простира от асистент по "Структури от данни в програмирането", "Обектно-ориентирано програмиране със C++" и "Visual C++" до лектор в курсовете "Мрежова сигурност", "Сигурен програмен код", "Интернет програмиране с Java", "Конструиране на качествен програмен код", "Програмиране за платформа .NET" и "Разработка на приложения с Java". Интересите на Николай са концентрирани върху техническата и бизнес страната на информационната сигурност, Java и .NET технологиите и моделирането и управлението на бизнес процеси в големи организации. Николай има бакалавърска степен от Факултета по математика и информатика на Софийски университет "Св. Климент Охридски". Като ученик е дългогодишен състезател по програмиране, с редица призови отличия. През 2004 г. е награден от Президента на България Георги Първанов за приноса му към развитието на информационните технологии и информационното общество. Той е почетен член на БАРС. Личният му сайт е достъпен от адрес: www.nedyalkov.com.

Панайот Добриков

Панайот Добриков е софтуерен архитект в SAP A.G., Java Server Technology (www.sap.com), Германия и е отговорен за координацията на софтуерните разработки в SAP Labs България. Той е завършил Факултета по математика и информатика на Софийски университет "Св. Климент Охридски", специалност информатика. Панайот е дългогодишен участник (като състезател и ръководител) в ученически и студентски състезания по програмиране и е носител на много престижни награди в страната и чужбина. Той е автор на книгите "Програмиране = ++Алгоритми;" (www.algoplus.org) и "Java Programming with SAP Web Application Server", както и на десетки научно-технически публикации. През периода 2001-2003 води курсовете "Проектиране и анализ на компютърни алгоритми" и "Прагматика на обектното програмиране" в Софийски университет. Можете да се свържете с Панайот по e-mail: dobrikov@gmail.com.

Преслав Наков

Преслав Наков е аспирант по изкуствен интелект в Калифорнийския университет в Бъркли (www.berkeley.edu), САЩ. Неговият професионален опит включва шестгодишна работа като софтуерен разработчик във фирмите Комсофт (www.comsoft.bg) и Рила Солюшънс (www.rila.bg). Интересите му са в областта на компютърната лингвистика и биоинформатиката. Преслав получава магистърската си степен по информатика от Софийски университет "Св. Климент Охридски". Той е носител е на бронзов медал от Балканиада по информатика, заемал призови места в десетки национални състезания по програмиране като ученик и студент. Състезател е, а покъсно и треньор на отбора на Софийския университет, участник в Световното междууниверситетско състезание по програмиране (ACM International Collegiate Programming Contest). Той е асистент в множество курсове във

Факултета по математика и информатика на Софийски университет, лектор-основател на курсовете "Проектиране и анализ на компютърни алгоритми" и "Моделиране на данни и проектиране на бази от данни". Преслав е автор на книгите "Основи на компютърните алгоритми" и "Програмиране = ++Алгоритми;" (www.algoplus.org). Той има десетки научни и научнопопулярни публикации в престижни международни и национални издания. Той е първият носител на наградата "Джон Атанасов" за принос към развитието на информационните технологии и информационното общество, учредена от президента на България Георги Първанов.

Радослав Иванов

Радослав Иванов е софтуерен разработчик във фирма Медсофт (www.medsoft.biz) и студент в специалност информатика във Факултета по математика и информатика на Софийски университет "Св. Климент Охридски". Професионалните му интереси са в областта на информационната сигурност и продуктите и технологиите на Microsoft.

Рослан Борисов

Рослан Борисов е софтуерен инженер във фирма Сирма Груп (www.sirma.bg), звено на Сирма Бизнес Консултинг. Професионалните му интереси са свързани основно с изграждане на приложения, базирани на технологии на Microsoft. Специализирал е в областта на билинг системи, като и основни и сателитни банкови системи. Има сериозен опит с платформата .NET Framework и сървърите за бази от данни Microsoft SQL Server и Oracle. Участва в различни проекти, свързани с български и чужди банки. В момента Рослан е студент в Нов български университет, специалност информатика. Можете да се свържете с него на e-mail: rosborisov@gmail.com.

Светлин Наков

Светлин Наков е директор на направление "обучение" на Националната академия по разработка на софтуер (http://academy.devbg.org), където обучава софтуерни специалисти за практическа работа в ИТ индустрията с Java и .NET платформите. Той е хоноруван преподавател по съвременни софтуерни технологии в Софийски университет "Св. Климент Охридски", където води курсове по "Проектиране и анализ на компютърни алгоритми", "Интернет програмиране с Java", "Мрежова сигурност", "Програмиране за .NET Framework", "Качествен програмен код" и "Разработка на уеб приложения с Java". Светлин има сериозен професионален опит като софтуерен разработчик и консултант. Неговите интереси обхващат Java технологиите, .NET платформата и информационната сигурност. Той е завършил бакалавърската и магистърската си степен във Факултета по математика и информатика на Софийски университет "Св. Климент Охридски". Като ученик и студент Светлин е победител в десетки национални състезания по програмиране и е носител на 4 медала от международни олимпиади по информатика. Той има десетки научни и

технически публикации, свързани с разработката на софтуер, в български и чуждестранни списания и е автор на книгите "Интернет програмиране с Java", "Java за цифрово подписване на документи в уеб" и ръководител на двата тома на настоящата книга. През 2003 г. той е носител на наградата "Джон Атанасов" на фондация Еврика. През 2004 г. получава награда "Джон Атанасов" от президента на България Георги Първанов за приноса му към развитието на информационните технологии и информационното общество. Светлин е един от учредителите на Българската асоциация на разработчиците на софтуер (www.devbg.org) и понастоящем неин председател.

Стефан Добрев

Стефан Добрев е старши уеб разработчик във фирма Vizibility (www.vizibility.net). Той отговаря за голяма част от .NET продуктите, разработвани в софтуерната компания, в това число уеб базирана система за изграждане на динамични сайтове и управление на тяхното съдържание, уеб система за управление на контакти и др. Негова отговорност е и внедряването на утвърдените практики и методологии за разработка на софтуер в производствения процес. Професионалните му интереси са насочени към уеб технологиите, в частност ASP.NET, XML уеб услугите и цялостната разработка на приложения, базирани на .NET Framework. Стефан следва информатика във Факултета по математика и информатика на Софийски университет "Св. Климент Охридски".

Стефан Кирязов

Стефан Кирязов е софтуерен разработчик във фирма Верео Технолъджис (www.vereo.bg). Той се занимава професионално с разработка на .NET решения за бизнеса и държавната администрация. Опитът му включва изграждане на уеб и настолни приложения с технологии на Microsoft, а също и Java и Oracle. Завършил е Факултета по математика и информатика на Софийски университет "Св. Климент Охридски", специалност информатика. Неговите професионални интереси включват архитектура, дизайн и методологии за разработка на големи корпоративни приложения. За контакти със Стефан можете да използвате неговия e-mail: stefan.kiryazov@gmail.com.

Стефан Захариев

Стефан Захариев работи като софтуерен разработчик в Интерконсулт България (www.icb.bg), където е отговорен за създаването на инструменти за автоматизиране на процеса на разработка. Той има дългогодишен опит в създаването на ERP системи, който натрупва при работата си в различни фирми в България. Основните му интереси са свързани със системите за управление на бази от данни, платформата .NET, ORM инструментите, J2ME, както и Borland Delphi. При завършването си на средното образование в "Технологично училище – Електронни системи", печели отличителна награда за цялостни постижения. През 2005 г. завършва "Технически

университет – София", където се дипломира като бакалавър във факултета по "Компютърни системи и управление". Той членува в БАРС и в Софийската .NET потребителска група Можете да се свържете със Стефан по e-mail: stephan.zahariev@gmail.com.

Стоян Дамов

Стоян Дамов е софтуерен консултант, пич, поет и революционер. Можете да се свържете с него по e-mail: stoyan.damov@gmail.com или от неговия личен сайт: http://spaces.msn.com/members/stoyan/.

Тодор Колев

Тодор Колев е софтуерен разработчик в Gugga (www.gugga.com) и студент във Факултета по математика и информатика на Софийски университет "Св. Климент Охридски", специалност Информатика. Неговите професионални интереси са в областта на обектно-ориентирания анализ, моделиране и програмиране, уеб технологиите, базите данни и RIA (Rich Internet Applications). Тодор е дългогодишен участник в състезания по информатика и информационни технологии, печелил редица грамоти и отличия, както и сребърен медал на международна олимпиада по информационни технологии. Той е носител на първо място от националната олимпиада по информационни технологии и е участвал в журито на същата олимпиада година по-късно. Тодор има множество разработки в сферата на уеб технологиите и е участвал в изследователски екип в Масачузетският технологичен институт (MIT). Той е сертифициран Microsoft специалист.

Христо Дешев

Христо Дешев е разработчик на ASP.NET компоненти във фирма telerik (www.telerik.com). Той е завършил Американския университет в България, специалност информатика. Основните му интереси са в областта на подобряването на процеса на разработка на софтуер. Той е запален привърженик на Agile методологиите, основно на Extreme Programming (XP). Професионалният му опит е предимно в разработката на решения с кратък цикъл за обратна връзка, високо покритие от тестове и почти пълна автоматизация на всички нива от работния процес.

Христо Радков

Христо Радков е управител на фирма за софтуерни консултантски услуги Calisto ID (www.calistoid.com). Той е бакалавър от английската специалност "Manufacturing Engineering" в Технически Университет – София и магистър по информационни и комуникационни технологии във Факултета по математика и информатика на Софийски университет "Св. Климент Охридски". Притежава сертификационна степен от Microsoft - MCSD.NET. Христо има дългогодишен опит с различни сървъри за бази от данни и сериозен опит с различни технологии на Microsoft, Borland, Sun и Oracle.

Участник и ръководител е в проекти за изграждане на няколко големи информационни системи, динамични Интернет портали и др. Под негово ръководство е създаден най-успешния складово-счетоводен софтуер за фармацевтични предприятия в страната. Като ученик Христо има множество участия и награди от олимпиади по математика в страната и чужбина.

Цветелин Андреев

Цветелин Андреев е софтуерен инженер във фирма Dreamix Ltd. (www.dreamix.eu). Той е член на Българската асоциация на разработчиците на софтуер и е инструктор към Националната академия по разработка на софтуер. Цветелин участва като лектор в редица курсове и семинари. Изявява се и като консултант по използване на модерни уеб технологии. Част от интересите му са свързани с платформата FreeBSD, в частност използването ѝ за разработка на софтуер. Член е на групата на българските потребители на FreeBSD (freebsd-bg.org). Цветелин е завършил бакалавърска степен по информатика във Факултета по математика и информатика на Софийски университет "Св. Климент Охридски", а сега е студент по Стопанско Управление в същия университет. Личният му уеб сайт е достъпен от адрес: www.flowerlin.net.

Явор Ташев

Явор Ташев е софтуерен разработчик във фирма ComMetric (www.commetric.com). Той е завършил Факултета по математика и информатика на Софийски университет "Св. Климент Охридски", специалност информатика. Участвал е в разработката на големи корпоративни сайтове, комуникационни системи и решения за обработка на статистически данни и прогнозиране с методи на изкуствен интелект, използвайки технологиите и платформите на Microsoft. Интересите му са насочени към .NET платформата, Java и изкуствения интелект. Професионалният му опит е свързан предимно с .NET Framework, Visual Studio .NET, Microsoft SQL Server и Microsoft Internet Information Server.

Благодарности

Настоящата книга стана реалност благодарение на много хора и няколко организации, които помогнаха и допринесоха за проекта. Нека изкажем своята благодарност и уважение към тях.

Светлин Наков

На първо място трябва да благодарим на главния организатор и ръководител на проекта, Светлин Наков, който успя да мотивира над 30 души да участват в начинанието и успя да ги ръководи успешно през всичките месеци на работата по проекта. Той успя да реализира своята идея за създаване на чисто българска книга за програмиране с .NET Framework най-вече благодарение на всички доб-

роволни участници, които дариха своя труд за проекта и отделиха от малкото си свободно време за да споделят своите знания и опит безвъзмездно, за каузата.

Авторският колектив

Авторският колектив е наистина главният виновник за съществуването на тази книга. Текст с такъв обем и такова качество не може да бъде написан от един или двама автора за по-малко от няколко години, а до тогава информацията може вече да остаряла.

Идеята за участие на толкова много автори се оказа успешна, макар и координацията между тях да не беше лесна. Въпреки, че отделните глави от книгата са писани от различни автори, те следват единен стил и високо качество. Всички глави са добре структурирани, с много заглавия и подзаглавия, с много и подходящи примери, с добър стил на изказ и еднакво форматиране.

Българска асоциация на разработчиците на софтуер

Проектът получи силна подкрепа от Българската асоциация на разработчиците на софтуер (БАРС), тъй като е в синхрон с нейните цели и идеи.

БАРС официално държи правата за издаване и разпространение на книгата в хартиен вид, но няма право да реализира печалба от тази дейност. Асоциацията чрез своите контакти успя да намери финансиране за отпечатването на книгата, както и хостинг за нейния уеб сайт и форум.

Microsoft Research

В ранните си фази, когато бяха изготвени лекциите за курса "Програмиране за .NET Framework", проектът получи подкрепа и частично финансиране от Microsoft Research. Ако не беше тази подкрепа, вероятно нямаше да се стигне до създаването на лекциите и до написването на книгата.

SciForge.org

Порталът за организиране на работата в екип SciForge.org даде своя принос към проекта, като предостави среда за съвместна работа, включваща система за контрол над версиите, форум, пощенски списък (mailing list) и някои други средства за улеснение на работата.

Благодарностите са отправени главно към създателя на портала и негов главен администратор Калин Наков (www.kalinnakov.com), който указваше редовно съдействие в случай на технически проблеми.

Софийски университет "Св. Климент Охридски"

Факултетът по математика и информатика (ФМИ) на Софийски университет "Св. Климент Охридски" подпомогна проекта главно в началната му фаза, като подкрепи предложението на преподавателския екип от курса "Програмиране за платформа .NET" за участие в конкурса на Microsoft Research. По-късно факултетът продължи да подкрепя инициативите на авторския колектив на книгата като им позволи да провеждат изборни курсове по програмиране за .NET Framework 1.1 и 2.0 за студентите от Софийски университет.

telerik

Софтуерната компания **t**elerik (www.telerik.com) подкрепи проекта чрез осигуряване на финансиране за отпечатване на книгата на хартия. Изказваме благодарности от името на целия авторски колектив.

Сайтът на книгата

Официалният уеб сайт на книгата "Програмиране за .NET Framework" е достъпен от адрес: http://www.devbg.org/dotnetbook/. От него можете да изтеглите цялата книга в електронен вид, лекциите, на които тя е базирана, както и сорс кода на практическия проект от глава 29, за който има специално изготвена инсталираща програма.

Към книгата е създаден и дискусионен форум, който се намира на адрес: http://www.devbg.org/forum/index.php?showforum=30. В него можете да дискутирате всякакви технически и други проблеми, свързани с книгата, да отправяте мнения и коментари и да задавате въпроси към авторите.

Лиценз

Книгата и учебните материали към нея се разпространяват свободно по следния лиценз:

Общи дефиниции

1. Настоящият лиценз дефинира условията за използване и разпространение на комплект учебни материали и книга по "Програмиране за .NET Framework", разработени от екип под ръководството на Светлин Наков (www.nakov.com) с подкрепата на Българска асоциация на разработчиците на софтуер (www.devbg.org) и Microsoft Research (research.microsoft.com).
2. Учебните материали се състоят от:
   - презентации;
   - примерен сорс код;
   - демонстрационни програми;

- задачи за упражнения;
- книга (учебник) по програмиране за .NET Framework с езика C#.

3. Учебните материали са достъпни за свободно изтегляне при условията на настоящия лиценз от официалния сайт на проекта:

   http://www.devbg.org/dotnetbook/

4. Автори на учебните материали са лицата, взели участие в тяхното изработване. Всеки автор притежава права само над продуктите на своя труд.
5. Потребител на учебните материали е всеки, който по някакъв начин използва тези материали или части от тях.

Права и ограничения на потребителите

1. Потребителите **имат** право:
   - да използват учебните материали или части от тях за всякакви цели, включително да ги да променят според своите нужди и да ги използват при извършване на комерсиална дейност;
   - да използват сорс кода от примерите и демонстрациите, включени към учебните материали или техни модификации, за всякакви нужди, включително и в комерсиални софтуерни продукти;
   - да разпространяват безплатно непроменени копия на учебните материали в електронен или хартиен вид;
   - да разпространяват безплатно оригинални или променени части от учебните материали, но само при изричното споменаване на източника и авторите на съответния текст, програмен код или друг материал.
2. Потребителите **нямат** право:
   - да разпространяват срещу заплащане учебните материали или части от тях (включително модифицирани версии), като изключение прави само програмният код;
   - да премахват настоящия лиценз от учебните материали.

Права и ограничения на авторите

1. Всеки автор притежава неизключителни права върху продуктите на своя труд, с които взима участие в изработката на учебните материали.
2. Авторите имат право да използват частите, изработени от тях, за всякакви цели, включително да ги изменят и разпространяват срещу заплащане.

3. Правата върху учебните материали, изработени в съавторство, са притежание на всички съавтори заедно.
4. Авторите нямат право да разпространяват срещу заплащане учебни материали или части от тях, изработени в съавторство, без изричното съгласие на всички съавтори.

Права и ограничения на БАРС

Ръководството на Българска асоциация на разработчиците на софтуер (БАРС) има право да разпространява учебните материали или части от тях (включително модифицирани) безплатно или срещу заплащане, но без да реализира печалба от продажби.

Права и ограничения на Microsoft Research

Microsoft Research има право да разпространява учебните материали или части от тях по всякакъв начин – безплатно или срещу заплащане, но без да реализира печалба от продажби.

Светлин Наков,
01.11.2006 г.

Глава 15. Изграждане на графичен потребителски интерфейс с Windows Forms

Автори

Светлин Наков

Радослав Иванов

Необходими знания

- Базови познания за .NET Framework
- Базови познания за езика C#
- Базови познания за делегатите и събитията в .NET Framework
- Начални умения за работа с Visual Studio .NET и Windows Forms редактора му

Съдържание

- Какво е Windows Forms?
- Програмни компоненти. Компонентен модел на .NET
- Програмен модел на Windows Forms. Модел на пречертаване на контролите
- Основни класове. Йерархия на класовете
- Класът **`Control`**. Други базови контроли
- Форми, прозорци и диалози – класът **`Form`**
- Основни контроли – **`TextBox`**, **`Label`**, **`Button`**
- Поставяне на контроли във формата
- Управление на събитията
- Windows Forms редакторът на VS.NET
- Стандартни диалогови кутии
- Извикване на диалогови кутии
- Други Windows Forms контроли. Менюта. Ленти с инструменти. Статус ленти
- Диалог за избор на файл

- MDI приложения
- Валидация на данни
- Свързване на данни (Data Binding). Навигация с **`CurrencyManager`**
- Контролата **`DataGrid`**
- Master-Details навигация
- Наследяване на форми
- Пакетът **`System.Drawing`** и GDI+
- Печатане на принтер
- Потребителски контроли
- Хостинг на контроли в Internet Explorer
- Нишки и Windows Forms
- Влачене (Drag and Drop)
- Конфигурационен файл на приложението

В тази тема ...

В настоящата тема ще разгледаме средствата на Windows Forms за създаване на прозоречно-базиран графичен потребителски интерфейс (GUI) за .NET приложенията. Ще се запознаем с програмния модел на Windows Forms, неговите базови контроли, средствата за създаване на прозорци, диалози, менюта, ленти с инструменти и статус ленти, както и с някои по-сложни концепции: MDI приложения, data-binding, наследяване на форми, хостинг на контроли в Internet Explorer, работа с нишки в Windows Forms и др.

Какво е Windows Forms?

Windows Forms е стандартната библиотека на .NET Framework за изграждане на прозоречно-базиран графичен потребителски интерфейс (GUI) за настолни (desktop) приложения. Windows Forms дефинира набор от класове и типове, позволяващи изграждане на прозорци и диалози с графични контроли в тях, чрез които се извършва интерактивно взаимодействие с потребителя.

При настолните приложения графичният потребителски интерфейс позволява потребителят директно да взаимодейства с програмата чрез мишката и клавиатурата, а програмата прихваща неговите действия и ги обработва по подходящ начин.

Windows Forms е базирана на RAD концепцията

В .NET Framework и особено в Windows Forms се поддържа концепцията за Rapid Application Development (RAD).

Какво е RAD?

RAD е подход за разработка, при който приложенията се създават визуално чрез сглобяване на готови компоненти посредством помощници и инструменти за автоматично генериране на голяма част от кода. В резултат приложенията се разработват много бързо, с малко ръчно писане на код и с намалени усилия от страна на програмиста.

При компонентно-ориентираната разработка всеки компонент решава някаква определена задача, която е част от проекта. Компонентите се поставят в приложението, след което се интегрират един с друг чрез настройка на техните свойства и събития. Свойствата на всеки компонент определят различни негови характеристики, а събитията служат за управление на действията, които са предизвикани от него.

Windows Forms позволява бърза визуална разработка

Windows Forms е типична компонентно-ориентирана библиотека за създаване на GUI, която предоставя възможност с малко писане на програмен код да се създава гъвкав графичен потребителски интерфейс.

Windows Forms позволява създаването на формите и другите елементи от графичния интерфейс на приложенията да се извършва визуално и интуитивно чрез подходящи редактори, като например Windows Forms Designer във Visual Studio .NET. По-нататък в настоящата тема ще разгледаме по-подробно конкретните възможности, които VS.NET предоставя за създаване на Windows Forms приложения.

Windows Forms и другите библиотеки за изграждане на GUI

Windows Forms прилича на много други библиотеки за изграждане на графичен потребителски интерфейс (GUI), но и сериозно се различава от повечето от тях.

Windows Forms и VCL

На идейно ниво Windows Forms много прилича на библиотеката Visual Component Library (VCL) от Delphi. Приличат си в голяма степен дори самите контроли, техните имена, свойства и събития. Това вероятно се дължи до голяма степен на участието на главния архитект на Delphi Андерс Хейлсбърг в разработката на Windows Forms и .NET Framework.

Windows Forms и Visual Basic 6

По начина на разработка Windows Forms прилича много и на Visual Basic 6, който позволява визуално изграждане на интерфейса, чрез влачене на компоненти и настройка на свойства и събития, също както в Delphi.

Windows Forms и MFC

По своята мощ Windows Forms не отстъпва на по-старите средства за изграждане на GUI, например MFC (Microsoft Foundation Classes) библиотеката, която се използваше във Visual C++ преди Microsoft да вземат стратегическото решение разработката на GUI за Windows да преминава постепенно към .NET Framework и Windows Forms.

За разлика от MFC, при Windows Forms, интерфейсът се изгражда няколко пъти по-бързо, по-лесно и почти без да се пише програмен код.

Windows Forms и Java AWT/Swing

AWT и Swing са библиотеки за изграждане на прозоречно-базиран GUI, които се използват при Java платформата. Програмният модел на Windows Forms има съществени разлики от програмния модел на AWT и Swing и причините за това произхождат най-вече от факта, че AWT и Swing са преносими библиотеки, предназначени да работят на много операционни системи, докато Windows Forms е базирана на Win32 API.

Контролите в Windows Forms

Windows Forms съдържа богат набор от стандартни контроли: форми, диалози, бутони, контроли за избор, текстови полета, менюта, ленти с инструменти, статус ленти и много други. В допълнение към стандартните контроли Windows Forms позволява на разработчиците по лесен начин да създават допълнително собствени контроли, които да използват като части в приложенията си.

В Интернет могат да се намерят безплатно или срещу лицензна такса голям брой библиотеки от контроли, които решават често срещани проб-

леми и спестяват време на разработчика при реализацията на често срещани задачи. Съществуват дори цели софтуерни компании, които професионално се занимават с производството на компоненти и контроли (като Infragistics, ComponentOne и българската **t**elerik).

Windows Forms и работа с данни

Windows Forms предоставя много контроли за визуализация и редактиране на данни – текстови, списъчни и таблични. За спестяване на време на разработчика е въведена концепцията "свързване на данни" (data binding), която позволява автоматично свързване на данните с контролите за тяхната визуализация. Ще обърнем специално внимание на концепцията "data binding" по-късно в настоящата тема.

Вградена поддръжка на Unicode

В Windows Forms поддръжката на Unicode е вградена. Всички контроли са съобразени с Unicode стандарта и позволяват използване на много езици и азбуки (латиница, кирилица, гръцки, арабски и др.) без допълнителни настройки на Windows или на приложението.

Наследяване на форми и контроли

Windows Forms е проектирана така, че да позволява лесно наследяване и разширяване на форми и контроли. Това дава възможност за преизползване на общите части на потребителския интерфейс. По-нататък в настоящата тема ще демонстрираме как точно се реализира това.

ActiveX контроли

Преди появата на .NET Framework Windows приложенията са били базирани на програмния модел "Win32". В Win32 среда се използват т. нар. ActiveX контроли, които се реализират чрез компонентния модел на Windows (COM – Component Object Model).

ActiveX контролите представляват графични компоненти. Те имат свойства, чрез които им се задават различни характеристики, и събития, управляващи поведението им.

ActiveX контролите много приличат на Windows Forms контролите от .NET Framework, но за разлика от тях се реализират с неуправляван код и преди използване трябва да се регистрират чрез добавяне в регистрите на Windows (Windows Registry).

Поради дългия период на развитие на Win32 платформата, има изключително много ActiveX контроли, които са създадени с течение на годините от различни софтуерни производители.

В .NET Framework по лесен начин, без да се пише ръчно програмен код, могат да се използват вече разработени ActiveX контроли. Например можем да вградим уеб браузъра Internet Explorer или четеца на PDF

документи Adobe Acrobat Reader като част от наше приложение. Как точно се използват ActiveX контроли в Windows Forms ще разгледаме в темата "Взаимодействие с неуправляван код".

Печатане на принтер

В Windows Forms са предоставени удобни средства за печатане на документи на принтер. Те предоставят достъп до всички стандартни диалози за печат, чрез които потребителите избират печатащо устройство и настройват неговите характеристики. Самото печатане се извършва със стандартните средства на .NET Framework за чертане върху повърхности.

Windows Forms контроли в Internet Explorer

При проектирането на .NET Framework е заложено Windows Forms контролите да могат да се изпълняват в средата на Internet Explorer или други уеб браузъри, без да се застрашава сигурността на потребителя.

Тази технология е една добра съвременна алтернатива на Java аплетите и позволява разширяване на функционалността на уеб приложенията с гъвкав интерактивен потребителски интерфейс. На практика се дава възможност .NET приложения да се изпълняват в браузъра на клиента като се вградят в най-обикновена уеб страница (подобно на Flash технологията).

Силна поддръжка на графика (GDI+)

Библиотеката Windows Forms широко използва средствата на Windows платформата за чертане и работа с графични обекти (GDI+). Windows Forms позволява тези средства да се използват за създаване на собствени изображения върху различни повърхности – в прозорец, върху принтер, плотер и др. Дава се достъп до всички по-важни примитиви за чертане – текст, графични изображения, геометрични фигури (точки, линии, правоъгълници, елипси) и т. н.

Нашето първо Windows Forms приложение

За да илюстрираме как се използва на практика Windows Forms, да разгледаме следното просто приложение:

```
using System;
using System.Windows.Forms;

public class SampleForm : System.Windows.Forms.Form
{
   static void Main()
   {
      SampleForm sampleForm = new SampleForm();
      sampleForm.Text = "Sample Form";
      Button button = new Button();
      button.Text = "Close";
```

```
      button.Click +=
        new EventHandler(sampleForm.button_Click);
      sampleForm.Controls.Add(button);
      sampleForm.ShowDialog();
      sampleForm.Dispose();
   }

   private void button_Click(object sender, EventArgs e)
   {
      Close();
   }
}
```

В него се създава прозорец, който съдържа бутон с текст "Close". При натискане на бутона прозорецът се затваря (това се реализира чрез прихващане и обработка на събитието "натискане на бутона").

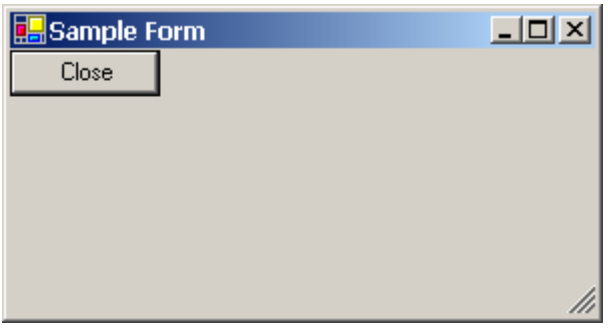
Как да компилираме и стартираме примера?

За да компилираме горното приложение, можем да ползваме конзолния компилатор на .NET Framework за езика C#:

```
csc SampleForm.cs
```

Можем да компилираме примера и от VS.NET, но за целта трябва да създадем нов Windows Application проект и да копираме кода в него.

При изпълнение на приложението се получава следния резултат:



Как работи примерът?

Нашето първо Windows Forms приложение е доста просто. То е изградено по следния начин:

- Дефиниран е клас **SampleForm**, който наследява класа **System.Windows.Forms.Form**. Този клас представлява главната форма на приложението.
- В главния метод **Main()** първо се задава заглавие за формата. След това се създава бутон, който се добавя в списъка с контролите на формата и се прихваща събитието "щракване върху бутона". Накрая

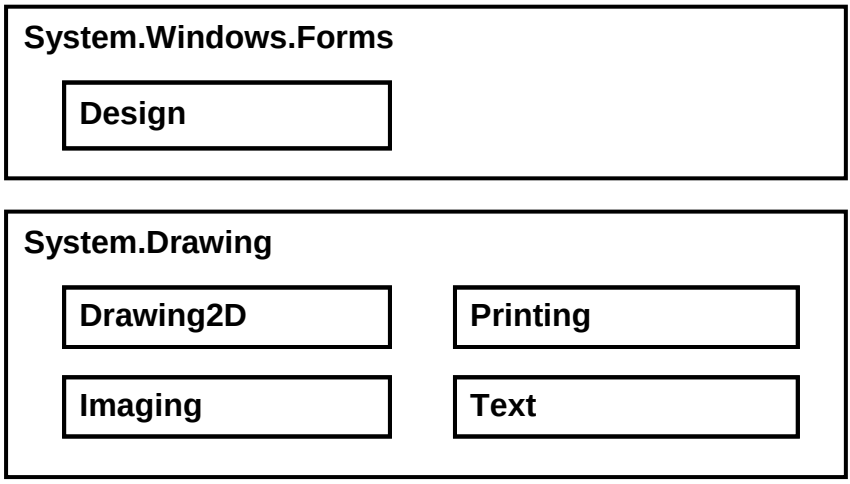
формата се показва в модален режим (модален режим означава, че другите форми на приложението не са активни, докато не се затвори текущата) и след затварянето й се унищожава.

- При натискане на бутона се извиква събитие, което затваря формата, и приложението завършва.

Примерът е доста прост и показва основните моменти при изграждането на потребителски интерфейс с Windows Forms – създаване на форми, поставяне на контроли във формите, настройка на свойствата на контролите, прихващане и обработване на събития.

Библиотеките на .NET за изграждане на GUI

Средствата на .NET Framework за изграждане на графичен потребителски интерфейс са дефинирани в пространствата от имена **`System.Drawing`** и **`System.Windows.Forms`**, които са реализирани съответно в асемблитата **`System.Drawing.dll`** и **`System.Windows.Forms.dll`**. Тези пространства заедно с пространствата, съдържащи се в тях, са изобразени на фигурата:



Пространството System.Windows.Forms

Класовете и типовете от пространството **`System.Windows.Forms`** осигуряват средства за работа с прозорци, диалози, контроли за въвеждане на текст, контроли за избор, менюта, ленти с инструменти, таблици, дървета и др.

Пространството System.Windows.Forms.Design

Пространството **`System.Windows.Forms.Design`** съдържа класове, които поддържат конфигурирането на компонентите и дефинират поведението на Windows Forms контролите по време на дизайн.

Пространството System.Drawing

Класовете и типовете от пространството **`System.Drawing`** и неговите подпространства осигуряват достъп до GDI+ функциите на Windows: работа с повърхности, точки, линии, четки, моливи, геометрични фигури, картинки, текст и шрифтове и др.

Програмни компоненти

В софтуерното инженерство компонентите са преизползваеми (reusable) програмни единици (класове), които решават специфична задача. Всеки компонент има ясно дефиниран интерфейс, който описва неговите свойства, методи и събития. Компонентите се използват като части от други компоненти или програми – те са градивните елементи на софтуера.

Компонентен модел

В софтуерното инженерство компонентният модел дефинира стандартите за разработка и използване на програмните компоненти и техния жизнен цикъл. Тези стандарти описват чрез интерфейси модела на поведение и взаимодействие на всички компоненти в дадена среда.

Компонентният модел на .NET Framework

Компонентният модел на .NET Framework дефинира програмния модел (система от правила) за създаване и използване на .NET компоненти. Този програмен модел се реализира чрез определени класове и интерфейси, които поддържат описанието на компонентите.

В .NET Framework компонентният модел позволява дефиниране на поведението на компонентите по време на дизайн (design-time behavior) и по време на работа (runtime behavior).

Компоненти и контейнери

В .NET Framework са дефинирани два вида преизползваеми обекти: компоненти и контейнери. Компонентите са функционални единици, които решават някаква задача, а контейнерите са обекти, които съдържат списък от компоненти.

Преизползваемост на компонентите

Благодарение на междуезиковата съвместимост, която CLR осигурява, .NET компонентите могат директно да се преизползват във всички .NET езици за програмиране. Възможно е .NET компоненти да бъдат използвани и от Win32 приложения, но за целта трябва да се публикуват във вид на COM обекти.

Пространството System.ComponentModel

Компоненти се използват не само в Windows Forms, а навсякъде в .NET Framework. По тази причина основната функционалност на компонентния модел на .NET се намира в пространството **`System.ComponentModel`**. В него са дефинирани основните интерфейси **`IComponent`** и **`IContainer`** и техните имплементации **`Component`** и **`Container`**.

Windows Forms и компонентният модел на .NET

В архитектурата на Windows Forms залягат концепциите на компонентния модел на .NET Framework. Компонентният модел на .NET дефинира компоненти и контейнери. По подобен начин Windows Forms дефинира контроли и контейнер-контроли.

Контроли и контейнер-контроли

Контролите в Windows Forms са всички компоненти, които са видими за потребителя (имат графично изображение). Те биват два вида: контейнер контроли (форми, диалози, панели и т.н.) и контроли (бутони, текстови полета, етикети, списъчни контроли и т.н.). Контейнерите са предназначени да съдържат в себе си други контроли (включително и други контейнер контроли), докато контролите са предназначени да се съдържат в контейнер контролите.

В Windows Forms всяка контрола може да се използва като контейнер-контрола, но за някои контроли това е безсмислено. Няма смисъл и не е правилно в бутон да се поставят други бутони или текстови полета.

Програмен модел на Windows Forms

Програмният модел на Windows Forms дефинира класовете за работа с форми, диалози и контроли, събитията на контролите, жизнения цикъл на приложенията, модела на пречертаване на контролите, модела на получаване и обработка на събитията и модела на управление на фокуса. Нека разгледаме всички тези елементи от програмния модел.

Форми

Windows Forms предлага стандартни класове за работа с форми (това са прозорците и диалозите в GUI приложенията). Формите могат да бъдат модални и немодални (по една или по много активни едновременно). Формите са контейнер-контроли и могат да съдържат други контроли, например етикети, текстови полета, бутони и т.н. Базов клас за всички форми е класът **`System.Windows.Forms.Form`**.

Контроли

Контролите в Windows Forms са текстовите полета, етикетите, бутоните, списъците, дърветата, таблиците, менютата, лентите с инструменти, статус лентите и много други. Windows Forms дефинира базови класове за контролите и класове-наследници за всяка контрола. Базов клас за всички контроли е класът **`System.Windows.Forms.Control`**. Пример за контрола е например бутонът (класът **`System.Windows.Forms.Button`**).

Събития

Всички контроли от Windows Forms дефинират събития, които програмистът може да прихваща. Например контролата **`Button`** дефинира събитието **`Click`**, което се активира при натискане на бутона. Събитията в Windows Forms управляват взаимодействието между програмата и контролите и между самите контроли.

Жизнен цикъл на Windows Forms приложенията

Жизненият цикъл на GUI приложенията е базиран на съобщения. Графичната среда на операционната система прихваща всички потребителски действия (напр. движението на мишката, натискането на клавиши от клавиатурата и т.н.) и ги натрупва в специална опашка. След това всяко съобщение се предава към приложението, за което се отнася и по-точно към нишката (thread) от приложението, за която се отнася.

Нишки и многозадачност

В многозадачните операционни системи (каквито са например Windows и Linux) е възможно едно приложение да изпълнява няколко задачи паралелно, като използва няколко нишки (threads) в рамките на процеса, в който работи програмата.

За целите на настоящата тема можем да си мислим, че нишките са нещо като отделни задачи в програмата, които се изпълняват едновременно (паралелно) в даден момент. По-нататък, в темата "Многонишково програмиране и синхронизация", ще обърнем специално внимание на многозадачността, използването и синхронизацията на нишки.

Опашката от събития

Всяка нишка от всяко приложение си има своя собствена опашка, в която постъпват съобщенията за всички събития, идващи от потребителя или от други източници. Всяко съобщение носи информация за събитието, което е настъпило – часът на настъпване, идентификатор на прозорец, за който се отнася събитието, тип на събитието, параметри на събитието (напр. номер на натиснатия клавиш при събитие от клавиатурата или позиция на курсора при събитие от мишката) и т.н. В Windows Forms съобщенията са инстанции на структурата **`System.Windows.Forms.Message`**.

Главната нишка на всяко Windows Forms приложение извършва една единствена задача: в безкраен цикъл обработва опашката от съобщения за приложението и предава постъпилите съобщения на контролата, за която са предназначени.

В Windows Forms приложенията винаги имат точно една нишка, която обработва всички съобщения, идващи от графичните контроли, и това е главната нишка на приложението. Графичният потребителски интерфейс на цялото приложение се управлява от тази нишка. При настъпване на събитие, свързано с някоя от формите на приложението или контролите в нея, в опашката на главната нишка постъпва съответно съобщение и то се обработва, когато му дойде редът.

Само главната нишка трябва да взаимодейства с опашката от събития

Много е важно, когато разработваме Windows Forms приложения, да се съобразяваме със следното правило:

Графичният потребителски интерфейс на приложението трябва да се управлява само и единствено от неговата главна нишка.

Ако не спазваме това правило, ще се сблъскаме с много странни и неприятни проблеми. Например, ако стартираме едновременно няколко нишки и от всяка от тях от време на време променяме съдържанието на определено текстово поле, е възможно в дадени моменти приложението да "зависва".

Всяка контрола обработва собствените си събития

Когато главната нишка на Windows Forms приложение получи съобщение, свързано с някоя от неговите форми, тя препраща съобщението до обработчика на съобщения на съответната форма. Този обработчик от своя страна проверява дали съобщението е за самата форма или за някоя нейна контрола. Ако съобщението е за формата, то се обработва директно от съответния обработчик на събития. Ако съобщението е за някоя от контролите във формата, то се предава на нея. Контролата, която получи съобщението, може да е обикновена контрола или контейнер-контрола. Когато обикновена контрола получи съобщение, тя го обработва директно. Когато контейнер-контрола получи съобщение, тя проверява дали то е за нея или е за някоя от вложените контроли. Процесът продължава, докато съобщението достигне до контролата, за която е предназначено.

По описаната схема всяко съобщение преминава от главната нишка на приложението през формата, за която се отнася, и евентуално през още една или няколко други контроли, докато си намери обработчика.

Обработка на събитие – пример

Нека имаме някакво приложение, което се състои от една форма, в която има един бутон. Да предположим, че натиснем левия бутон на мишката, докато курсорът е върху бутона във формата. Какво се случва?

Главната нишка на приложението получава съобщение "натиснат ляв бутон на мишка", в което са записани координатите, в които е бил курсорът на мишката в момента на натискането. Операционната система подава тези координати относително спрямо горния ляв ъгъл на формата.

Докато обработва съобщението, главната нишка на приложението открива формата, за която се отнася събитието (това е най-горната от всички форми, в които попада курсорът на мишката) и го предава на нейния обработчик на събития.

Формата получава съобщението и вижда, че то се отнася за някаква позиция, в която се намира някаква нейна контрола (в случая това е бутонът). Формата преценява, че съобщението не е за нея, а е за бутона, и му го предава.

Бутонът получава събитието и вижда, че то е предназначено точно за него. Събитието бива погълнато (консумирано) от обработчика на събития на бутона и съответно бутонът преминава в състояние "натиснат". Самият бутон малко след това изпраща събитие за пречертаване до самия себе си (на пречертаването ще обърнем внимание след малко). Когато това събитие достигне по същия път до бутона, той се пречертава в натиснато състояние.

Прекратяване на Windows Forms приложение

При затваряне на главната форма на Windows Forms приложение, към нея се изпраща съобщение за затваряне. Формата се затваря в момента, в който получи съобщението и го обработи. В резултат на затварянето на формата се прекратява цикълът, в който главната нишка на приложението обработва пристигащите за нея съобщения и приложението приключва изпълнението си.

Модел на пречертаване на контролите

В Windows Forms контролите често се пречертават, например при преместване на прозорец, при смяна на активния прозорец или при промяна на размера, позицията или състоянието на някоя контрола. При всяко от изброените действия една или няколко контроли, които попадат в обсега на даден засегнат регион, се обявяват за невалидни и се активира процесът на пречертаване.

Процесът на пречертаване

Процесът на пречертаване на контрола, която е засегната от промяна в нея самата, от промяна на контейнер-контролата, в която се намира, или от промяна в други съседни контроли, се извършва на два етапа:

1. За контролата се извиква методът **`Invalidate()`**, който обявява за невалидна дадената контрола или отделен неин участък и изпраща заявка за пречертаване. **`Invalidate()`** реално маркира регионите от контролата, които по някаква причина имат нужда от пречертаване и след това й изпраща съобщение "пречертай" (**`WM_PAINT`**), което се изпълнява по-късно.
2. В някакъв момент цикълът за обработка на съобщения на текущата нишка получава съобщението "пречертай" и в резултат изпълнява метода **`Paint()`** на съответната контрола. Този метод извършва самото графично обновяване на всички невалидни участъци от контролата или в частност я пречертава цялата.

Друг интересен метод, свързан с пречертаването на контролите, е **`Update()`** методът. Той може да се използва след **`Invalidate()`** за незабавно пречертаване на дадена контрола чрез насилствено извикване на **`Paint()`**, без да се изчаква **`Paint()`** да бъде извикан от цикъла за обработка на съобщения за текущата нишка.

Съобщението "пречертай"

Съобщението "пречертай" (**`WM_PAINT`**) е специално съобщение. То се обработва последно, едва след като всички останали съобщения от опашката на главната нишка вече са обработени и в нея останат само съобщения "пречертай". Това осигурява намаляване на претрепванията на контролите, когато те се променят много пъти за кратко време.

Например, ако при обработката на дадено събитие на дадена контрола бъде изпратено 5 пъти съобщение "пречертай", контролата ще изпълни само едно пречертаване и то едва след като формата е обработила всички останали съобщения и е станало ясно кои контроли в момента са невалидни и трябва да се пречертаят.

Реалното графично изобразяване на заявените за пречертаване контроли се извършва, когато те обработват съобщението "пречертай", което може да е много след като пречертаването е заявено.

Когато се пречертават няколко контроли последователно, те винаги се пречертават в реда, в който контролите са поставени в контейнер-контролата (т. нар. Z-order). Първи се пречертават най-рано поставените контроли, а последни – най-късно поставените.

Реализация на пречертаването

Всяка Windows Forms контрола може да дефинира програмен код, който реализира изчертаването на нейното съдържание (метод **`Paint()`**).

Windows Forms контролите могат да се поставят една върху друга със застъпване. Понеже при пречертаване контролите се изобразяват една след друга по реда на поставянето им, ако има застъпвания, последно поставената контрола закрива (частично или напълно) всички контроли, с които се застъпва.

По-нататък в настоящата тема ще дадем примерен код, който реализира пречертаването на контрола чрез използване на графичните примитиви от GDI+.

Управление на фокуса и навигация

В една форма в даден момент може някоя от контролите да е активна, т.е. да държи фокуса. Контролата, която е на фокус, обикновено показва това по някакъв начин – бутонът променя графичния си вид, текстовото поле показва мигащ курсор и т.н.

При настъпване на събитие от клавиатурата, то се получава първо от контролата, която е на фокус. Например, ако едно текстово поле е на фокус и потребителят натисне клавиш, който съответства на някоя буква, текстовото поле обикновено приема буквата и я изписва на позицията на курсора. Ако текстовото поле не обработи натиснатия клавиш (например, ако това е клавиш за навигация **[Tab]**), той се обработва от контейнер-контролата.

Windows Forms осигурява навигация между контролите чрез клавишите **[Tab]** и **[Shift+Tab]**, които преместват фокуса към следващата или предходната контрола. Коя е следващата и коя е предишната контрола се определя от т. нар. "Tab Order", който зависи от реда на поставяне на контролите във формата и от някои свойства на контролите.

Формите също могат да са на фокус (да са активни) или да не са. Фокусът между формите може да се променя от потребителя само при немодални форми. Модалните форми не позволяват друга форма да приема фокуса, докато не бъдат затворени.

Текущата фокусирана контрола и форма могат да се променят, както в резултат от потребителски действия от клавиатурата и мишката, така и програмно - чрез изпращане на подходящи съобщения или извикване на подходящи методи. Има контроли, които не могат да приемат фокуса, и контроли, които могат да го приемат, но се прескачат при натискане на **[Tab]** и **[Shift+Tab]**.

Основни класове в Windows Forms

Библиотеката Windows Forms дефинира съвкупност от базови класове за контролите, контейнер-контролите, както и множество графични контроли и неграфични компоненти.

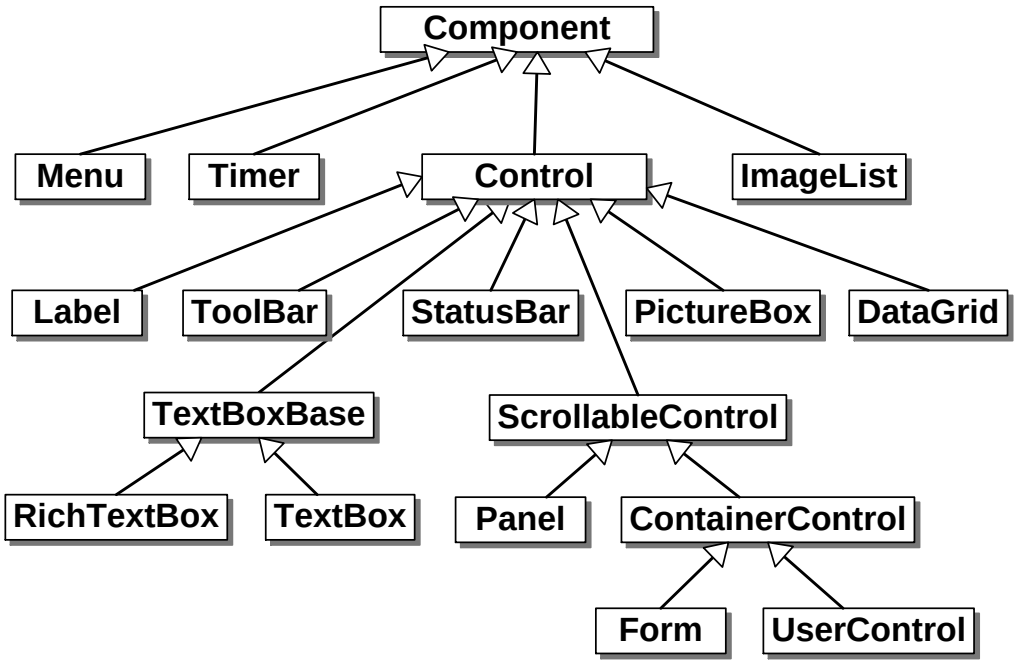
Основните базови класове, използвани в Windows Forms, са:

- **System.ComponentModel.Component** – представлява .NET компонент. Използва се за реализацията на неграфични компоненти. Например компонентата **System.Windows.Forms.Timer** е наследник на класа **Component**.

- **System.Windows.Forms.Control** – представлява графична контрола. Графични контроли са компонентите, които имат графичен образ. Всички Windows Forms контроли са наследници на класа **Control**, включително и контейнер-контролите.
- **System.Windows.Forms.ScrollableControl** – представлява контрола, която поддържа скролиране на съдържанието си. Може да съдържа в себе си други контроли.
- **System.Windows.Forms.ContainerControl** – представлява контрола, която съдържа в себе си други контроли и осигурява управление на фокуса. Не всички контейнер-контроли наследяват този клас. Например панелът (**System.Windows.Forms.Panel**) може да съдържа в себе си други контроли, но е наследник на класа **ScrollableControl**, а не на **ContainerControl**.

Йерархия на класовете

На клас-диаграмата по-долу е показана част от класовата йерархия на библиотеката Windows Forms:



Забелязва се, че не всички класове от Windows Forms са контроли. Някои са обикновени .NET компоненти, например **Menu**, **Timer** и **ImageList**. Изглежда малко странно защо менюто не е контрола, но това е така, защото компонентата **Menu** реално няма графичен образ и представлява списък от **MenuItem** елементи. **MenuItem** класът вече има графичен образ и следователно е контрола.

Типичните контроли (**Label**, **TextBox**, **Button**, **ToolBar**, **StatusBar** и др.) са наследници на класа **Control**. Общото за всички тях е, че имат графичен образ и се управляват чрез съобщения.

Контролите, които могат да се скролират (например панелите) са наследници на **ScrollableControl**. Контролите, които съдържат други контроли и се грижат за управление на фокуса (например формите и диалозите), наследяват **ContainerControl**.

Класът Control

Класът **System.Windows.Forms.Control** заема много централна роля в библиотеката Windows Forms. Той е базов клас, основа за всички графични контроли, и определя единна рамка за контролите – програмен модел, по който да се разработват и изпълняват. В него са дефинирани общите за всички контроли свойства и събития.

Свойства на класа Control

Нека сега разгледаме по-важните свойства на класа **Control**:

- **Anchor**, **Dock** – задават по какъв начин контролата се "закотвя" за контейнера си. Тези свойства са много полезни, ако искаме да управляваме размерите и позицията на контролата при промяна на размерите на контейнера, в който е поставена. Например чрез свойството **Anchor** можем да закотвим дадена контрола на определено разстояние от долния десен ъгъл на формата, в която стои, и при преоразмеряване това разстояние ще се запазва и контролата ще се движи заедно с движението на долния десен ъгъл на контейнера, в който е поставена.
- **Bounds** – задава размера (ширина и височина) и позицията на горния ляв ъгъл на контролата в рамките на нейния контейнер. Ако контролата е форма, позицията се задава спрямо горния ляв ъгъл на екрана. Ако контролата е елемент от форма (например бутон), позицията се отчита спрямо горния ляв ъгъл на формата (или контейнер-контролата), в която е оставена. Размерът включва цялото графично пространство на контролата. Например, ако контролата е форма, се включва и нейната рамка.
- **BackColor** – задава цвета на фона. Цветовете са инстанции на структурата **System.Drawing.Color**, която дефинира множество стандартни цветове и позволява потребителски дефинирани цветове, състоящи се от 4 на брой 8-битови компонента (яркост, червено, зелено и синьо).
- **ContextMenu** – задава контекстно меню (popup menu) за контролата. Контекстното меню обикновено се появява при натискане на десния бутон на мишката върху контролата.

- **Controls** – съдържа колекция от вложените в контролата други контроли (ако има такива). Например формите (инстанции на класа **Form**) съдържат в колекцията си **Controls** контролите, които са разположени в тях. По принцип всички Windows Forms контроли имат колекция **Controls** и могат да съхраняват в нея други контроли, но за някои от тях не е коректно това да се прави. Например не е коректно в бутон да поставяме друг бутон или текстово поле. Ако го направим, се появяват неприятни аномалии.

- **CanFocus** – връща дали контролата може да получава фокуса. Почти всички видове контроли могат да бъдат фокусирани, стига да не са забранени (**Enabled=false**).

- **Enabled** – позволява забраняване на контролата. Когато една контрола бъде забранена (**Enabled=false**), тя остава видима, но става неактивна. Обикновено забранените контроли се изобразяват с избледнял цвят, за да се различават от останалите. Забранените контроли не могат да получават фокуса. В частност забранен бутон не може да бъде натиснат, в забранено текстово поле не може да се пише и т.н. Ако забраним контейнер-контрола, която съдържа в себе си други контроли, всички тези контроли стават забранени.

- **Font** – задава шрифта, с който се изписва текстът в контролата (ако контролата по някакъв начин визуализира текст). При текстови полета това е шрифтът на текста в полето. При бутон това е шрифтът на текста в бутона. При етикет това е шрифтът на текста на етикета. Ако се зададе свойството **Font** за формата, всички контроли, които не дефинират изрично **Font**, го наследяват от формата. Шрифтът, с който е изобразено заглавието на формите, не може да се променя от Windows Forms. Той се настройва от графичната среда на операционната система (от контролния панел при Windows).

  Шрифтовете имат следните характеристики: наименование на шрифт (например **Arial**) или фамилия шрифтове (например **Monospace**, **SansSerif** или **Serif**), стил (например **Bold**, **Italic**, ...), размер (например 12 pt или 10 px) и кодова таблица (Cyrillic, Western, Greek, ...). Кодовата таблица е необходима рядко – само за старите шрифтове, които не поддържат Unicode.

- **ForeColor** – задава цвета на контролата.

- **Location** – съдържа позицията на контрола в нейния контейнер (координатите на горния й ляв ъгъл). За форми това е позицията на екрана, а за други контроли това е позицията във формата или контейнер-контролата.

- **Parent** – задава контейнер-контролата, в която се намира текущата контрола. Може и да няма такава (стойност **null**). Формите най-често имат стойност **null** за свойството **Parent**.

- **Size** – съдържа размерите на контролата (ширина и височина).

- **TabIndex** – определя реда при навигация с **[Tab]** и **[Shift+Tab]**.
- **TabStop** – задава дали контролата трябва да се фокусира при навигация с **[Tab]** и **[Shift+Tab]**. Ако се зададе **TabStop=false**, фокусът не спира в контролата при преминаване към следващата контрола (контролата се прескача).
- **Text** – задава текст, свързан с контролата. При етикет това е текстът, изобразен в етикета. При бутон това е текстът, изобразен в бутона. При текстово поле това е текстът, въведен в полето. При форма това е заглавието на формата. Текстът е в Unicode и това позволява да се използват свободно букви и знаци на латиница, кирилица, гръцки, арабски и други азбуки, стига избраният шрифт да съдържа съответните знаци.
- **Visible** – задава видимост на контролата. Ако за дадена контрола се зададе **Visible=false**, тя се скрива (изчезва, все едно не съществува). Скрита контрола може да се покаже отново, като й се зададе **Visible=true**.

Методи на класа Control

Публичните методи на класа **Control** се наследяват и са достъпни във всички Windows Forms контроли. По-важните от тях са:

- **Focus()** – фокусира контролата (ако е възможно).
- **Hide()**, **Show()** – скрива/показва контролата (ефектът е като да зададем **Visible=false** / **Visible=true**).

Събития на класа Control

Знаем колко са важни събитията за Windows Forms контролите. Благодарение на тях програмистът може да пише код, който се задейства при различни промени в състоянието на контролите. Ще разгледаме по-важните събития на класа **Control**:

- **Click** – настъпва при щракване с мишката върху контролата. При бутон това събитие се извиква при натискане на бутона. При форма **Click** се извиква при щракване с левия бутон на мишката върху формата, ако в съответната позиция няма друга контрола. Събитието не подава допълнителна информация в аргументите си.
- **Enter**, **Leave** – настъпват съответно при активиране и деактивиране на дадена контрола, т.е. когато контролата получи и загуби фокуса. При форми тези събития не се извикват.
- **KeyDown**, **KeyUp** – настъпват при натискане и отпускане на произволен клавиш (включително специалните клавиши като **[F1]**, **[Alt]**, **[Caps Lock]**, **[Start]** и др.). Събитието подава в аргументите си инстанция на класа **KeyEventArgs**, която съдържа информация за

натиснатия клавиш – име на клавиша (инстанция на изброения тип **System.Windows.Forms.Keys**) и информация за състоянието на клавишите **[Shift]**, **[Alt]** и **[Ctrl]**.

- **KeyPress** – настъпва при натискане на неспециален клавиш или комбинация от клавиши. Това събитие се активира само ако натиснатата клавишна комбинация се интерпретира като символ. Например натискането на клавиша **[Alt]** не води до получаване на символ и не задейства това събитие, докато натискането на клавиша **[V]** генерира някакъв символ в зависимост от текущия език. Събитието подава в аргументите си инстанция на **KeyPressEventArgs** класа, която съдържа символа, генериран в резултат от натискането на клавиша.
- **MouseDown**, **MouseMove**, **MouseUp**, **MouseWheel** – настъпват при събития от мишката, извършени върху контролата – натискане на бутон, движение на показалеца на мишката или преместване на колелото. Събитията подават в аргументите си инстанция на **MouseEventArgs** класа, която съдържа информация за състоянието на бутоните и колелото на мишката и за координатите на показалеца (изчислени спрямо горния ляв ъгъл на контролата).
- **MouseEnter**, **MouseLeave**, **MouseHover** – настъпват при навлизане, излизане и преместване на позицията на показалеца на мишката в рамките на контролата.
- **Move** – настъпва при преместване на контролата. Преместването може да се предизвика от потребителя (например преместване на форма) или програмно (чрез промяна на свойството **Location**).
- **Paint** – настъпва при пречертаване на контролата (при обработката на съобщението **WM_PAINT**). В това събитие контролата трябва да извърши пречертаването на графичния си образ. Събитието получава в аргументите си инстанция на **PaintEventArgs**, която съдържа **Graphics** обекта, върху който трябва да се извърши чертането.
- **Resize** – настъпва при промяна на размера на контролата. Може да се предизвика както от потребителя (при преоразмеряване на форма), така и програмно (при промяна на свойството **Size**).
- **TextChanged** – настъпва при промяна на свойството **Text** на контролата.
- **Validating** – използва се за валидация на данните, въведени в контролата. Валидацията на данни ще бъде дискутирана по-късно в настоящата тема.

Класът ScrollableControl

Класът **ScrollableControl** е наследник на класа **Control** и добавя към него функционалност за скролиране. Ето по-важните му свойства:

- **AutoScroll** – задава дали при нужда контролата ще получи автоматично скролиращи ленти.
- **HScroll**, **VScroll** – задават дали контролата да има хоризонтална и вертикална скролираща лента.

Класът ContainerControl

Класът **ContainerControl** осигурява функционалност за управление на фокуса. Свойството му **ActiveControl** съдържа във всеки един момент контролата, която е на фокус.

Форми, прозорци и диалози

Формите и диалозите в Windows Forms са прозорци, които съдържат контроли. Те могат да бъдат различни видове: да имат или нямат рамка, да са модални или не, да са разтегливи или не, да са над всички други прозорци или не и т.н.

Класът System.Windows.Forms.Form

Класът **System.Windows.Forms.Form** е базов клас за всички форми в Windows Forms GUI приложенията. Той представлява графична форма - прозорец или диалогова кутия, която съдържа в себе си контроли и управлява навигацията между тях.

Повечето прозорци имат рамка и специални бутони за затваряне, преместване и други стандартни операции. Външният вид на прозорците и стандартните контроли по тяхната рамка зависят от настройките на графичната среда на операционната система. Програмистът има само частичен контрол над външния вид на прозорците.

Класът **Form** е наследник на класовете **Control**, **ScrollableControl** и **ContainerControl** и наследява от тях цялата им функционалност, всичките им свойства, събития и методи.

По-важни свойства на класа Form

Всички прозорци и диалози в Windows Forms наследяват класа **Form** и придобиват от него следните свойства:

- **FormBorderStyle** – указва типа на рамката на формата. По-често използваните типове рамка са следните:
  - **Sizable** – стандартна разширяема рамка. Потребителят може да променя размерите на такива рамки.
  - **FixedDialog** – диалогова рамка с фиксирани размери. Такива рамки не могат да се преоразмеряват от потребителите.

  - **None** – липса на рамка. Цялото пространство на формата се използва за нейното съдържание.
  - **FixedToolWindow** – кутия с инструменти с фиксиран размер. Рамката не може да се преоразмерява от потребителите и е малко по-тясна от стандартната. Прозорци с такива рамки не се виждат в лентата на задачите (taskbar) на Windows Explorer и при натискане на **[Alt+Tab]**.

- **Controls** – съдържа списък с контролите, разположени във формата. От реда на контролите в този списък зависи редът, в който те се чертаят на екрана (Z-order) и редът, в който се преминава от една контрола към друга при навигация (tab order). Редът на преместване на фокуса може да се настройва и допълнително от свойствата **TabStop** и **TabIndex**.
- **Text** – заглавие на прозореца. Използва се Unicode, т.е. можем да използваме, кирилица, латиница, гръцки и други азбуки от Unicode стандарта.
- **Size** – размери на прозореца (ширина и височина). Включва цялото пространство, заемано от формата (рамката + вътрешността).
- **ClientSize** – размери на вътрешността на формата (без рамката й).
- **AcceptButton** – бутон по подразбиране. Този бутон се натиска автоматично, когато потребителят натисне клавиша **[Enter]**, независимо от това в коя контрола от формата е фокусът в този момент. Целта е да се улесни потребителя при попълването на форми с информация.
- **ActiveControl** – съдържа контролата, която държи фокуса. При промяна на това свойство се променя текущата фокусирана контрола.
- **ControlBox** – задава дали формата трябва да съдържа стандартните контроли за затваряне, минимизация и т. н.
- **Icon** – задава икона на прозореца.
- **KeyPreview** – ако се зададе **true**, позволява формата да обработва събитията от клавиатурата, преди да ги предаде на фокусираната контрола. Ако стойността е **false**, всяко събитие от клавиатурата се обработва само от контролата, която е на фокус.
- **MinimumSize**, **MaximumSize** – задава ограничения за размера на формата – максимална и минимална ширина и височина. При опит за преоразмеряване не се позволява потребителят да задава размер, който не е в тези граници.
- **Modal** – връща дали формата е модална. Когато една форма е модална, докато тя е активна, потребителят не може да работи с други форми от същото приложение. Всеки опит за преминаване в друга форма не успява, докато потребителят не затвори модалната форма.

Ако дадено приложение покаже едновременно няколко форми, които не са модални, потребителят ще може да преминава свободно между тях, без да ги затваря. Свойството **Modal** е само за четене. Модалността може да се задава първоначално, но не може да се променя, след като формата е вече показана.

- **Opacity** – задава прозрачност на формата (число от 0.00 до 1.00). Възможно е да не се поддържа или да работи много бавно при някои по-стари видеоадаптери.
- **MdiChildren** – в MDI режим извлича / задава подчинените форми на текущата форма. MDI (Multiple-Document Interface) е режим, при който дадена форма на приложението (обикновено главната форма) може да съдържа в себе си други форми, които са разположени в нейното работно пространство (като обикновени контроли).
- **MdiParent** – в MDI режим извлича / задава формата, която е собственик на текущата форма. Важи само за подчинени (child) форми.
- **TopMost** – задава дали формата стои над всички други прозорци (always on top). В такъв режим, дори ако формата не е активна, тя остава видима и стои над всички останали форми.
- **WindowState** – извлича състоянието на формата. Формата във всеки един момент е в някое от състоянията на изброения тип **FormWindowState** – нормално, минимизирано или максимизирано. По подразбиране формите са в нормално състояние – имат нормалния си размер. В максимизирано състояние формите временно променят размера си и заемат целия екран без лентата за задачи (task bar) на Windows Explorer. В минимизирано състояние формите са скрити и се виждат само в лентата за задачи (task bar).

По-важни методи на класа Form

Прозорците и диалозите в Windows Forms наследяват от класа **Form** следните базови методи:

- **Show()** – показва формата и я прави активна (фокусира я). Формата се показва в немодален режим. Извикването на този метод е еквивалентно на присвояването **Visible=true**. Изпълнението на този метод приключва веднага.
- **ShowDialog()** – показва формата в модален режим и след като тя бъде затворена, връща като резултат стойност от тип **DialogResult**. Тази стойност съдържа информация за причината за затваряне на формата. Изпълнението на метода **ShowDialog()** приключва едва след затваряне на формата, т.е. методът е блокиращ. По-нататък в настоящата тема ще обърнем специално внимание на извикването на модални форми и получаването на стойностите от контролите в тях.

- **Close()** – затваря формата. Когато една форма бъде затворена, тя изчезва и се освобождават използваните от нея ресурси. След като една форма бъде затворена, тя не може да бъде повече показвана. За временно скриване на форма трябва да се използва методът **Hide()**, а не **Close()**.
- **LayoutMdi(…)** – в MDI режим този метод пренарежда дъщерните (child) форми, съдържащи се в текущата форма. Начинът на пренареждане се задава от програмиста. Поддържат се няколко вида пренареждане - каскадно, хоризонтално, вертикално и др.

По-важни събития на класа Form

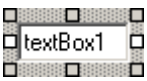
Всички прозорци и диалози в Windows Forms поддържат съвкупност от стандартни събития, които наследяват от класа **Form**:

- **Activated** / **Deactivate** – извикват се при активиране / деактивиране на формата (когато формата получи / загуби фокуса).
- **Closing** – извиква се при опит за затваряне на формата (например, когато потребителят натисне стандартния бутон за затваряне). Реализацията може да предизвиква отказване на затварянето. Събитието подава в аргументите си инстанция на класа **CancelEventArgs**, която има булево свойство **Cancel**, чрез което може да се откаже затварянето.
- **Load** – извиква се еднократно преди първото показване на формата. Може се ползва за инициализиране на състоянието на контролите.

Основни контроли в Windows Forms

Да разгледаме най-често използваните контроли в Windows Forms: **TextBox**, **Label** и **Button**.

TextBox



TextBox контролата е поле за въвеждане на текст. Може да бъде едноредово или многоредово. По-важните свойства на **TextBox** са:

- **Multiline** – задава дали контролата представлява само един ред или допуска въвеждането на няколко реда текст.
- **Text** – съдържа въведения в контролата текст. Когато свойството **Multiline** е **true**, за достъп до въведения текст може да се използва и свойството **Lines**.
- **Lines** – масив от символни низове, съдържащ въведения текст. Всеки елемент от масива съдържа един от редовете на текста.

Label

Контролата **Label** се използва за изобразяване на текст във формата. Свойството й **Text** съдържа текста, който се изобразява.

Button

Контролата **Button** представлява бутон, който може да бъде натискан. По-важни нейни свойства и събития са:

- **Click** – активира се при натискане на бутона.
- **Text** – задава текста, изобразяван върху бутона.

Поставяне на контроли във формата

Поставянето на контроли във форма става чрез добавянето им към колекцията от контроли на формата. Това може да се извърши чрез метода **Controls.Add(…)**:

```
Form form = new Form();
Button button = new Button();
button.Text = "Close";
form.Controls.Add(button);
```

Редът на контролите (т. нар. Z-order, който споменахме по-рано в тази тема) се определя от реда на поставянето им – последната контрола е най-отгоре. Когато използваме Windows Forms дизайнерът на Visual Studio .NET, той се грижи за правилното поставяне на контролите.

Управление на събитията

Прихващането на събитие става чрез добавянето на обработчик за него. За целта създаваме метод, който ще обработва събитието, и след това се абонираме за него. Ето пример:

```
Form form = new Form();
Button button = new Button();
button.Click += new EventHandler(this.button_Click);

...

private void button_Click(object sender, EventArgs e)
{
   // Handle the "click" event
```

```
}
```

Windows Forms дизайнерът на Visual Studio .NET улеснява прихващането на събития, като генерира автоматично обработчиците при избор на събитие от страницата "Events" на прозореца "Properties".

В Windows Forms има няколко типа събития:

- **`EventHandler`** – извършва проста нотификация, без да подава допълнителни данни за възникналото събитие.
- **`KeyEventHandler`** – събития от клавиатурата. Подава се информация кой е натиснатият клавиш, както и информация за състоянието на клавишите **`[Ctrl]`**, **`[Shift]`** и **`[Alt]`**.
- **`MouseEventHandler`** – събития от мишката. Подава се информация за позицията на мишката и състоянието на нейните бутони.
- **`CancelEventHandler`** – събития, които могат да откажат започнатото действие. Примерно, ако прихващаме събитието **`Closing`** на дадена форма, което е от тип **`CancelEventHandler`**, и потребителят се опита да затвори формата, можем да откажем затварянето, ако данните не са запазени.

Прост калкулатор – пример

Настоящият пример илюстрира използването на Windows Forms за създаването на просто приложение – калкулатор за събиране на цели числа:

```
using System;
using System.Drawing;
using System.Windows.Forms;

public class CalculatorForm : Form
{
  private TextBox TextBoxNumber1;
  private TextBox TextBoxNumber2;
  private TextBox TextBoxSum;
  private Button ButtonCalc;
  private Label LabelPlus;
  private Label LabelEquals;

  public CalculatorForm()
  {
    TextBoxNumber1 = new TextBox();
    TextBoxNumber1.Bounds = new Rectangle(
      new Point(16, 16), new Size(72, 20));
    TextBoxNumber1.MaxLength = 10;

    LabelPlus = new Label();
    LabelPlus.AutoSize = true;
```

```
    LabelPlus.Location = new Point(94, 19);
    LabelPlus.Text = "+";

    TextBoxNumber2 = new TextBox();
    TextBoxNumber2.Bounds = new Rectangle(
      new Point(112, 16), new Size(72, 20));
    TextBoxNumber2.MaxLength = 10;

    LabelEquals = new Label();
    LabelEquals.AutoSize = true;
    LabelEquals.Location = new Point(191, 18);
    LabelEquals.Text = "=";

    TextBoxSum = new TextBox();
    TextBoxSum.Bounds = new Rectangle(
      new Point(208, 16), new Size(72, 20));
    TextBoxSum.ReadOnly = true;

    ButtonCalc = new Button();
    ButtonCalc.Bounds = new Rectangle(
      new Point(16, 48), new Size(264, 23));
    ButtonCalc.Text = "Calculate sum";
    ButtonCalc.Click += new EventHandler(
      this.ButtonCalc_Click);

    this.AcceptButton = ButtonCalc;
    this.ClientSize = new Size(298, 87);
    this.Controls.Add(TextBoxNumber1);
    this.Controls.Add(LabelPlus);
    this.Controls.Add(TextBoxNumber2);
    this.Controls.Add(LabelEquals);
    this.Controls.Add(TextBoxSum);
    this.Controls.Add(ButtonCalc);
    this.FormBorderStyle = FormBorderStyle.FixedDialog;
    this.MaximizeBox = false;
    this.MinimizeBox = false;
    this.Text = "Calculator";
  }

  private void ButtonCalc_Click(object aSender, EventArgs aArgs)
  {
    try
    {
      int value1 = Int32.Parse(TextBoxNumber1.Text);
      int value2 = Int32.Parse(TextBoxNumber2.Text);
      int sum = value1 + value2;
      TextBoxSum.Text = sum.ToString();
    }
    catch (FormatException)
    {
```

```
            TextBoxSum.Text = "Invalid!";
        }

        TextBoxNumber1.SelectAll();
        TextBoxNumber2.SelectAll();

        TextBoxNumber1.Focus();
    }

    static void Main()
    {
        CalculatorForm CalcForm = new CalculatorForm();
        Application.Run(CalcForm);
    }
}
```

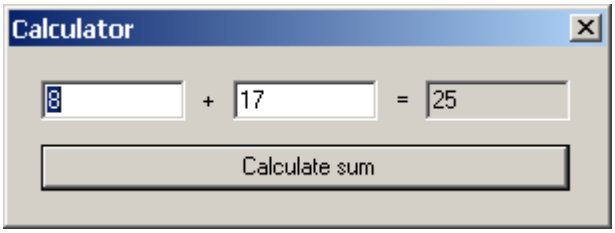
Как да компилираме и стартираме примера?

За да компилираме примера, можем да ползваме конзолния компилатор на .NET Framework за езика C#:

```
csc CalculatorForm.cs
```

Можем да извършим компилацията и от VS.NET, но за целта трябва да създадем нов Windows Application проект и да копираме кода в него.

Ето как изглежда примерното приложение в действие:



Как работи примерът?

В примера сме дефинирали класа **`CalculatorForm`**, който наследява класа **`System.Windows.Forms.Form`**. Този клас представлява главната форма на нашето приложение.

В класа дефинираме необходимите ни контроли – три **`TextBox`** контроли (две за въвеждане на числа и една за извеждане на сумата им), две **`Label`** контроли и един бутон, при натискането на който ще се изчислява резултатът от събирането на числата.

В конструктора на формата инициализираме контролите и ги добавяме в нея. За целта им задаваме размери, местоположение и някои други свойства. За текстовите полета, в които потребителят ще въвежда числата, които ще събираме, задаваме максималната им дължина в брой симво-

ли. За **Label** контролите задаваме текста, който ще визуализират. За бутона задаваме заглавие. Накрая задаваме начина, по който ще изглежда нашата форма.

В метода **CalcButton_Click(…)** обработваме събитието **Click** на бутона за изчисляване на сумата. В него парсваме съдържанието на двете текстови полета, сумираме числовите стойности, получени от тях, и записваме сумата в третото текстово поле. При грешка задаваме невалиден резултат.

Windows Forms редакторът на VS.NET

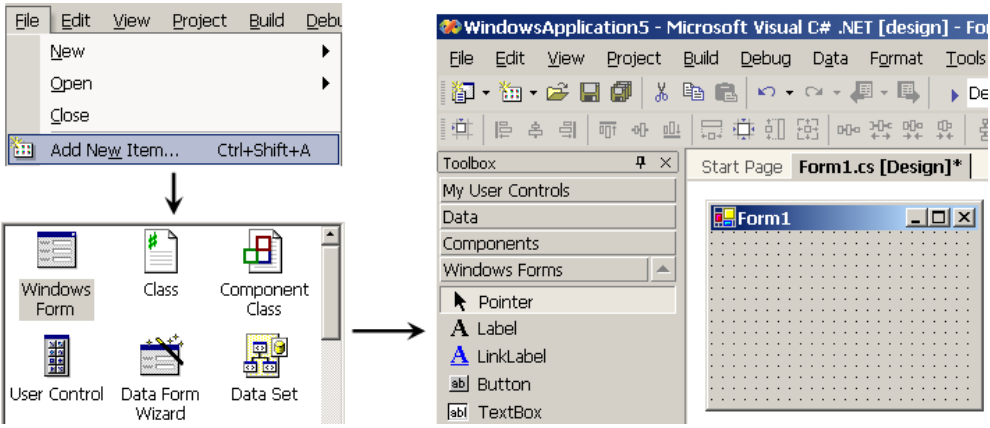
Създаването на форми, добавянето на контроли, настройката на размерите и местоположението на контролите и други такива операции, можем да извършваме, пишейки директно кода за нашето приложение, както в предходния пример. Разработката на приложения и създаването на потребителски интерфейс по този начин, обаче, е трудоемък и времеотнемащ процес.

Windows Forms редакторът на VS.NET ни дава възможност да правим всички тези неща визуално, ускорявайки процеса на разработка. Той улеснява значително извършването на следните операции:

- създаване на форми
- добавяне на контроли във формите
- добавяне на неграфични компоненти във формите
- настройка на свойствата на форми, компоненти и контроли
- добавяне на събития за форми, компоненти и контроли

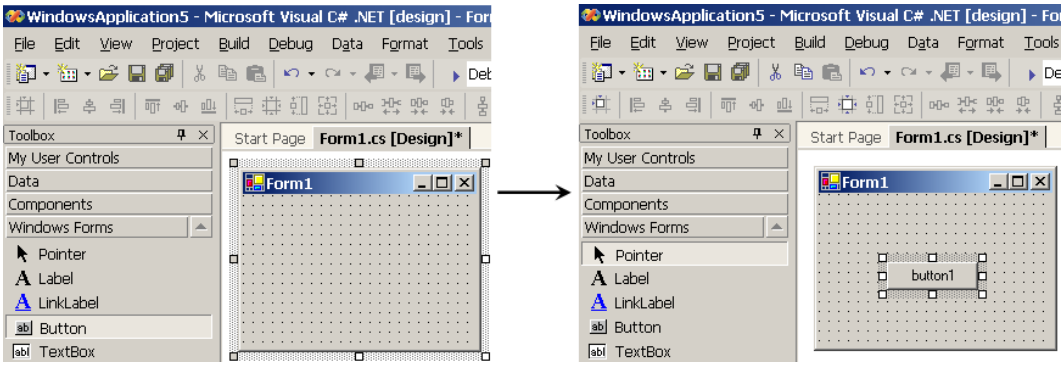
Създаване на форма

Създаването на форма във VS.NET става, като от менюто **File** изберем **Add New Item**. В появилия се диалогов прозорец избираме **Windows Form**, в полето за име въвеждаме името на формата и натискаме бутона **Open**. Нашата нова форма се отваря в редактора на VS.NET:



Добавяне на контрола

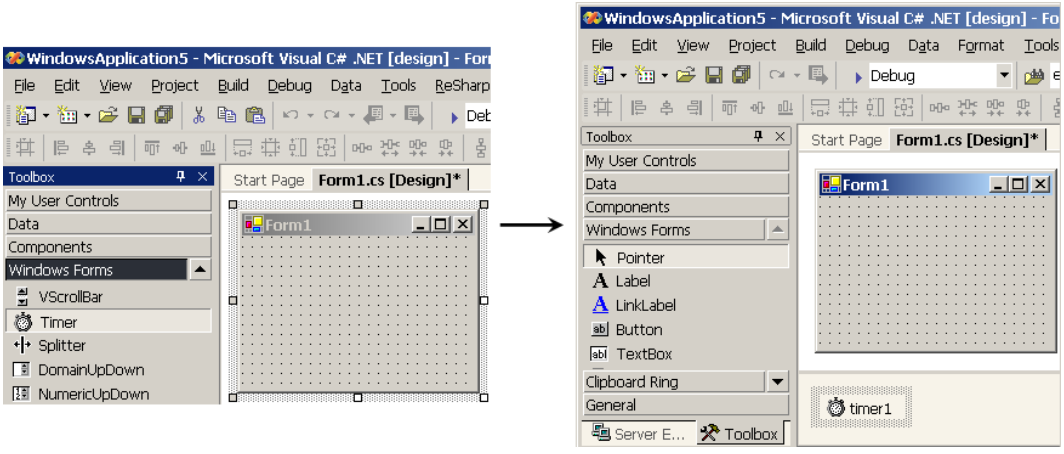
Добавянето на контрола става, като отворим формата, щракнем върху контролата в **Toolbox**, след това щракнем върху формата там, където искаме да е горният ляв ъгъл на контролата, и изтеглим мишката до там, където искаме да е долният й десен ъгъл. Контролата се добавя във формата с определеното местоположение и размери:



Всички контроли имат подразбиращ се размер. Ако желаем да добавим контрола с подразбиращия се размер, можем просто да я изтеглим от **Toolbox** и да я пуснем във формата (drag and drop).

Добавяне на неграфични компоненти

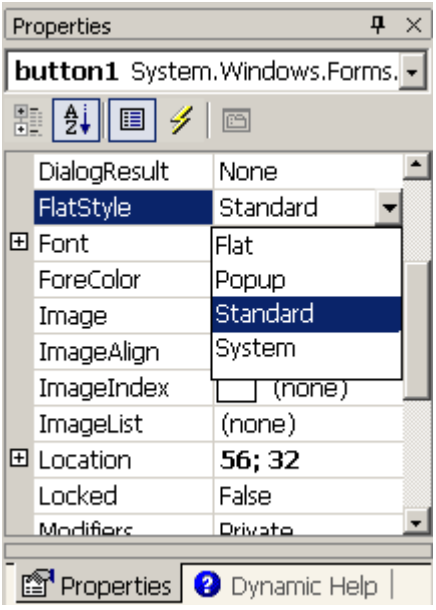
За да добавим неграфична компонента, отваряме формата, щракваме върху компонентата в **Toolbox** и я изтегляме върху формата. Тъй като неграфичните компоненти нямат потребителски интерфейс, те не се показват върху формата, а се изобразяват в специална област под нея:



Настройка на свойства

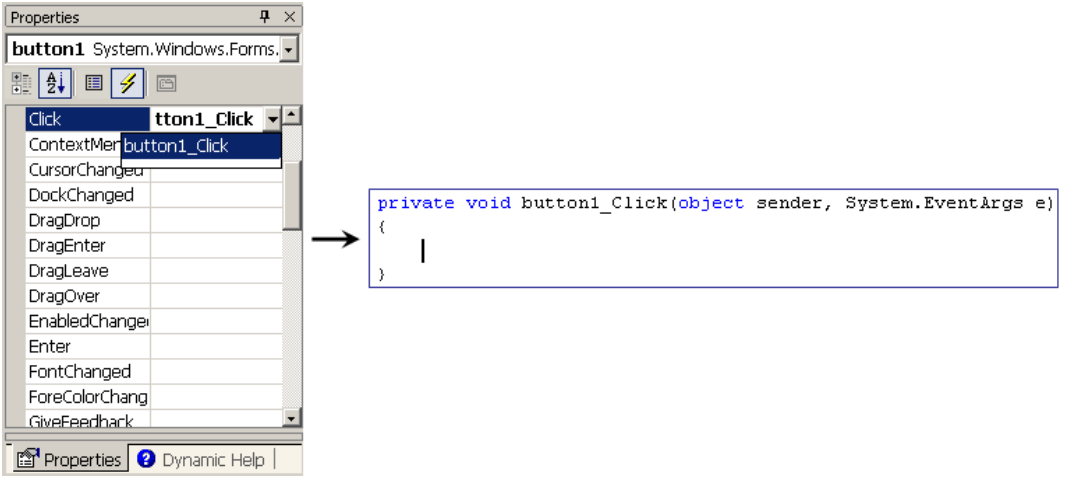
Настройката на свойства се извършва в прозореца **Properties** на редактора. Ако прозорецът не е видим, можем да го покажем, като

изберем **View | Properties Window** от менюто, натиснем **[F4]** или изберем **Properties** от контекстното меню, появяващо се при щракване с десния бутон на мишката върху контролата. От падащия списък, намиращ се най-отгоре в прозореца, избираме обекта, чиито свойства ще настройваме. След това избираме свойството, което ще променяме, и му задаваме стойност. В зависимост от свойството ще зададем текст, числова стойност или ще изберем стойността от списък. Ето как изглежда прозорецът **Properties** на VS.NET:



Добавяне на обработчици на събития

Добавянето на обработчици на събития също става от прозореца **Properties** на VS.NET:



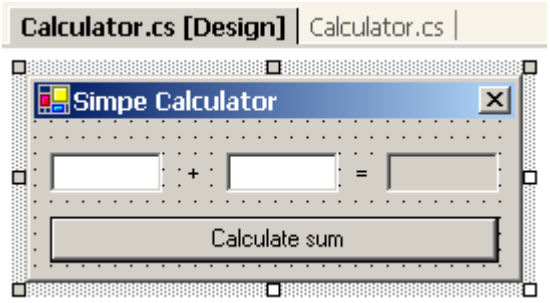
За целта от падащия списък, намиращ се най-отгоре в прозореца, избираме обекта, чиито свойства ще настройваме, и натискаме бутона **Events**, намиращ се под падащия списък. Появяват се събитията на обекта. От падащия списък срещу събитието, за което искаме да добавим обработчик, избираме метода, който ще обработва събитието. Ако ще дефинираме нов метод за обработка на събитието, изписваме неговото име в полето. Друга възможност е да щракнем 2 пъти с мишката и VS.NET ще избере име по подразбиране (името на контролата + "_" + името на събитието, примерно **OkButton_Click**). При създаване на обработчик за събитие Windows Forms редакторът добавя или намира метода и отваря редактора за код, позициониран точно върху него.

Създаване на калкулатор с Windows Forms редактора на VS.NET – пример

С настоящия пример ще илюстрираме използването на Windows Forms редактора на VS.NET за създаването на просто приложение – калкулатор, който събира цели числа. Функционалността на калкулатора ще е същата като на калкулатора от предишния пример, но този път ще използваме Windows Forms редактора, който ще генерира по-голямата част от кода на приложението.

Ето стъпките за създаването на нашия калкулатор:

1. Стартираме VS.NET и създаваме нов Windows Forms проект.
2. Задаваме на главната форма име **Calculator** и заглавие "**Simple Calculator**". Променяме и името на файла от **Form1.cs** на **Calculator.cs**.
3. Вземаме от Toolbox на VS.NET три **TextBox**, две **Label** и една **Button** контроли и ги поставяме в главната форма. Задаваме подходящи имена на поставените компоненти. Препоръчително е името на една контрола да съдържа нейното предназначение и тип (или префикс, указващ типа). В нашия случай подходящи имена са: **TextBoxNumber1**, **TextBoxNumber2**, **TextBoxSum**, **LabelPlus**, **LabelEquals** и **ButtonCalcSum**.
4. Задаваме празен низ в свойството **Text** на текстовите полета. За полето **TextBoxSum** задаваме **ReadOnly** да е **true**. На свойството **Text** на **ButtonCalcSum** задаваме стойност "**Calculate sum**". На свойствата **Text** на **LabelPlus** и **LabelEquals** задаваме съответно стойности "**+**" и "**=**". Ето как изглежда формата на нашия калкулатор в този момент:

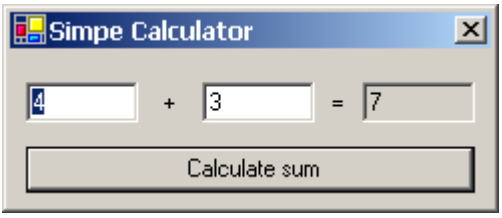


5. Остава да дефинираме събитието за натискане на бутона. С двойно щракване върху бутона VS.NET ни дава възможност да напишем кода за обработка на събитието му **`Click`**:

```
private void ButtonCalcSum_Click(object sender,
   System.EventArgs e)
{
   try
   {
      int value1 = Int32.Parse(TextBoxNumber1.Text);
      int value2 = Int32.Parse(TextBoxNumber2.Text);
      int sum = value1 + value2;
      TextBoxSum.Text = sum.ToString();
   }
   catch (FormatException)
   {
      TextBoxSum.Text = "Invalid!";
   }

   TextBoxNumber1.SelectAll();
   TextBoxNumber2.SelectAll();

   TextBoxNumber1.Focus();
}
```

При натискане на бутона парсваме съдържанието на двете текстови полета, сумираме числовите стойности, получени от тях, и записваме сумата в третото текстово поле. При грешка задаваме невалиден резултат.

6. Приложението вече е готово и можем да го стартираме и тестваме. Ето как изглежда нашият калкулатор:



Диалогови кутии

При разработката на Windows Forms приложения често пъти се налага да извеждаме диалогови кутии с някакви съобщения или с някакъв въпрос. Нека разгледаме стандартните средства за такива ситуации.

Стандартни диалогови кутии

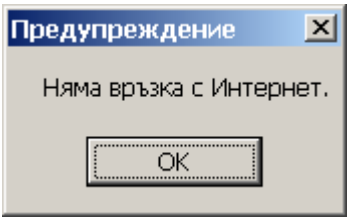
Класът **MessageBox** ни позволява да извеждаме стандартни диалогови кутии, съдържащи текст, бутони и икони:

- съобщения към потребителя
- въпросителни диалози

Показването на диалогова кутия се извършва чрез извикване на статичния метод **Show(…)** на класа **MessageBox**. Следният код, например, ще покаже диалогова кутия със заглавие "**Предупреждение**" и текст "**Няма връзка с интернет**":

```
MessageBox.Show("Няма връзка с Интернет.", "Предупреждение");
```

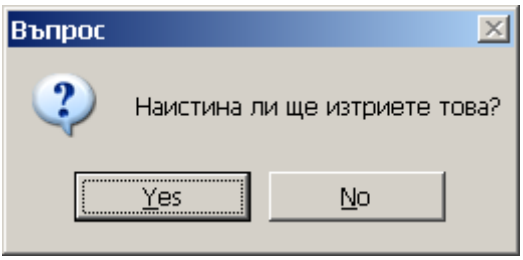
Ето как изглежда диалоговата кутия:



Нека разгледаме още един пример за стандартна диалогова кутия с малко повече функционалност:

```
bool confirmed =
  MessageBox.Show("Наистина ли ще изтриете това?",
  "Въпрос", MessageBoxButtons.YesNo,
  MessageBoxIcon.Question) == DialogResult.Yes;
```

Този код ще покаже диалогова кутия със заглавие "**Въпрос**" и текст "**Наистина ли ще изтриете това?**". Преди текста ще има икона с въпросителен знак в нея, а под него – бутони **Yes** и **No**. Ако потребителят натисне **Yes**, променливата **confirmed** ще има стойност **true**, в противен случай ще има стойност **false**. Ето как изглежда диалоговата кутия от примера:



Повече информация за класа **MessageBox** може да се намери в MSDN.

Извикване на диалогови кутии

Освен стандартните диалогови кутии можем да използваме и потребителски дефинирани диалогови кутии. Те представляват обикновени форми и се извикват модално по следния начин:

```
DialogResult result = dialog.ShowDialog();
```

Методът **ShowDialog()** показва формата като модална диалогова кутия. Типът **DialogResult** съдържа резултата (**OK**, **Yes**, **No**, **Cancel** и др.) от извикването на диалога. Задаването на **DialogResult** може да става автоматично, чрез свойството **DialogResult** на бутоните, или ръчно – преди затварянето на диалога чрез свойството му **DialogResult**.

Ако извиквате форма модално, след това задължително трябва да й извиквате `Dispose()` метода, за да освободите ресурсите, които тя е използвала. В противен случай те ще се освободят едва когато се активира Garbage Collector и ще се използват ненужно дълго.

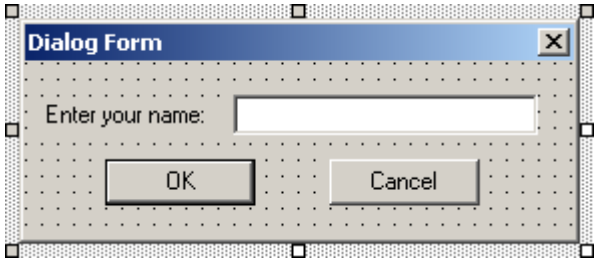
DialogResult и предаване на данни между диалози – пример

С настоящия пример ще илюстрираме използването на диалози в Windows Forms, ще покажем как диалозите могат да се извикват един друг и как могат да си предават данни.

В примера ще създадем един диалог, съдържащ текстово поле за въвеждане на име и два бутона – **OK** и **Cancel**. Този диалог ще се показва при натискане на бутон от главната форма. Ако потребителят въведе име и натисне **OK**, ще се показва диалог, съдържащ въведеното име, а ако потребителят затвори диалога, натискайки **Cancel**, ще се появи диалог, указващ, че е натиснат **Cancel**.

Ето и стъпките за изграждане на нашия пример:

1. Стартираме VS.NET и създаваме нов Windows Forms проект. В редактора се появява главната форма на приложението. На нея ще се спрем след малко.
2. Създаваме нова форма (**File | Add New Item … | Windows Form**). Сменяме името ѝ на **DialogForm**, а името на нейния файл – на **DialogForm.cs**. Задаваме на свойствата ѝ **MinimizeBox** и **MaximizeBox** стойности **false**, а на свойството **FormBorderStyle** стойност **FixedDialog**. Тази форма ще служи за въвеждане на името на потребителя.
3. Вземаме от Toolbox на VS.NET една **Label**, една **TextBox** и две **Button** контроли и ги подреждаме върху формата. Задаваме им подходящи имена. В нашия случай подходящи са имената: **LabelYourName**, **TextBoxName**, **ButtonOK** и **ButtonCancel**.
4. Задаваме свойството **Text** на **LabelYourName** да е "**Enter your name:**", на **ButtonOk** да е "**OK**", на **ButtonCancel** да е "**Cancel**", а на **TextBoxName** – празен низ.
5. Задаваме на свойството **DialogResult** на бутона **ButtonOk** стойност **OK**. По този начин при натискането му формата ще се затвори и ще бъде върнат резултат **DialogResult.OK**. Аналогично на свойството **DialogResult** на бутона **ButtonCancel** задаваме стойност **Cancel**. Ето как изглежда нашият диалог:



6. Остава да добавим на тази форма едно свойство **UserName**, което да извлича съдържанието на текстовото поле за въвеждане на потребителско име:

```
public string UserName
{
   get
   {
      return TextBoxName.Text;
   }
}
```

7. Поставяме върху главната форма бутон с име **ButtonCallDialog** и задаваме на свойството му **Text** стойност "**Call Dialog**". Чрез този бутон ще извикваме диалога, който създадохме по-рано.

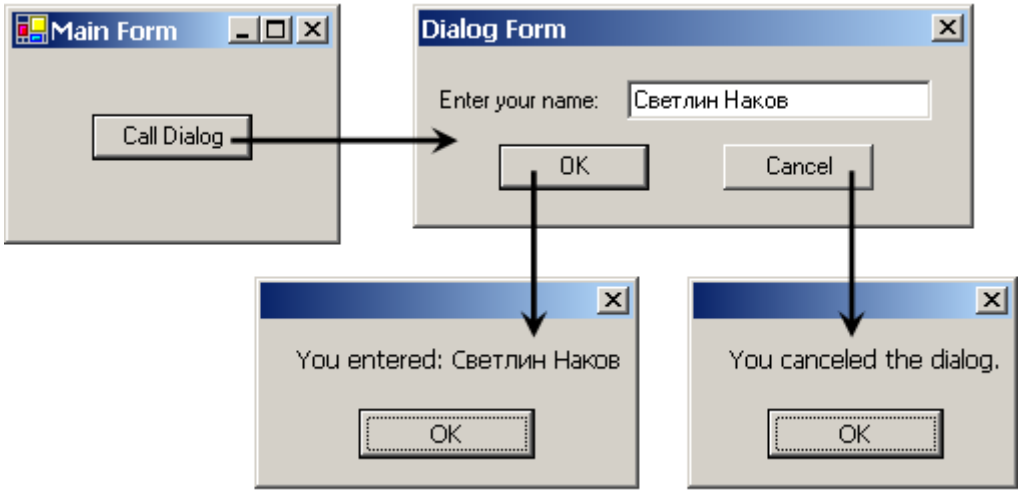
8. Добавяме обработчик на събитието **Click** на бутона:

```
private void ButtonCallDialog_Click(object sender,
   System.EventArgs e)
{
   DialogForm dialog = new DialogForm();
   if (dialog.ShowDialog() == DialogResult.OK)
   {
      string userName = dialog.UserName;
      MessageBox.Show("You entered: " + userName);
   }
   else
   {
      MessageBox.Show("You canceled the dialog.");
   }
   dialog.Dispose();
}
```

В него първо създаваме инстанция на **DialogForm**. След това извикваме модално формата **DialogForm** и проверяваме дали е била затворена с бутона **OK** чрез върнатия **DialogResult**. Ако е така, извличаме от **DialogForm** свойството **UserName**, с което взимаме въведеното в нея име и го показваме в диалогова кутия. Ако не е бил натиснат бутонът **OK**, това означава, че е бил натиснат бутонът **Cancel**. В този случай показваме диалогова кутия, указваща, че е натиснат бутон **Cancel**.

9. Задаваме на главната форма име **MainForm** и заглавие **"Main Form"**. Променяме и името на файла от **Form1.cs** на **MainForm.cs**.

10. Нашето приложение е готово и можем да го стартираме и тестваме:



Други Windows Forms контроли

Вече разгледахме най-основните контроли в Windows Forms – текстовите полета и бутоните. Нека сега разгледаме и някои други контроли, които също се използват често при изграждането на потребителски интерфейс.

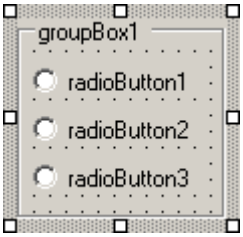
CheckBox

CheckBox е кутия за избор в стил "да/не". Свойството й **Checked** задава дали е избрана.

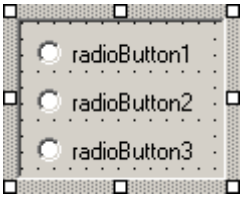
RadioButton

RadioButton е контрола за алтернативен избор. Тя се използва в групи. Всички **RadioButton** контроли в даден контейнер (например форма) образуват една група и в нея само един **RadioButton** е избран в даден момент.

За да създадем няколко групи в една форма, трябва да поставим всяка група в свой собствен контейнер, като например **GroupBox**, **Panel** или **TabPage**. Свойството **Checked** задава дали контролата е избрана. При промяна на **Checked** свойството се активира събитието **CheckedChanged**.

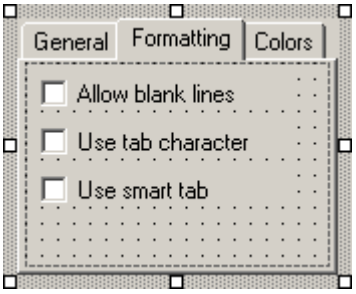


Panel



Panel представлява контейнер, който съдържа група други контроли. Служи за групиране на контроли. Когато преместим даден панел на друга позиция, всички контроли, които са в него, също се преместват. Ако стойността на свойството **Enabled** на **Panel** контролата има стойност **false**, то всички контроли, съдържащи се в нея, ще бъдат деактивирани.

TabControl и TabPage



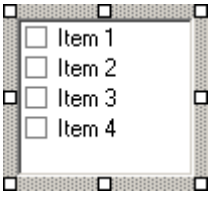
Контролите **TabControl** и **TabPage** осигуряват ползването на табове със страници. **TabControl** съдържа множество **TabPage** контроли, които се добавят в него чрез свойството **Controls**.

ListBox

ListBox контролата се използва за изобразяване на списък със символни низове, които потребителят може да избира чрез щракване с мишката върху тях. По-важните свойства на тази контрола са:

- **Items** – колекция, която задава списъка от елементи, съдържащи се в контролата.
- **SelectionMode** – разрешава/забранява избирането на няколко елемента едновременно.
- **SelectedIndex**, **SelectedItem**, **SelectedIndices**, **SelectedItems** – връщат избрания елемент (или избраните елементи).

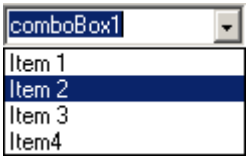
CheckedListBox



CheckedListBox изобразява списък от възможности за избор "да/не". По-важни свойства са:

- **Items** – задава възможностите, от които потребителят ще избира.
- **CheckedItems** – връща избраните елементи.

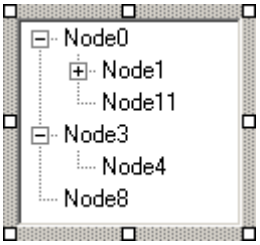
ComboBox



ComboBox представлява кутия за редакция на текст с възможност за drop-down алтернативен избор.

- **Text** – съдържа въведения текст.
- **Items** – задава възможните стойности, от които потребителят може да избира.
- **DropDownStyle** – задава стила на контролата – дали само се избира стойност от списъка или може да се въвежда ръчно и друга стойност.

TreeView



TreeView контролата изобразява дървовидни данни. Основни нейни свойства са:

- **Nodes** – съдържа дървото (списък от **TreeNode** елементи).
- **SelectedNode** – съдържа текущо избрания възел в дървото.

RichTextBox

RichTextBox е кутия за редакция на текст с форматиране (Rich Text Format). Методите **LoadFile(…)** и **SaveFile(…)** позволяват зареждане и записване на текста от контролата в Rich Text Format (RTF) файл или в текстов файл. Свойствата **SelectionStart** и **SelectionEnd** служат за извличане и задаване на областта от текста, която е маркирана. Чрез свойствата **SelectionFont**, **SelectionColor** и **SelectionAlignment** могат да се задават шрифт, цвят и подравняване на текущия маркиран текст.

LinkLabel

LinkLabel контролата е подобна на **Label** контролата, но изглежда като препратка (hyperlink). Свойството **Text** задава текстовото съдържание на контролата. При щракване с мишката върху препратката се активира събитието **LinkClicked**.

PictureBox

PictureBox се използва за изобразяване на картинки. Картинката, която ще се изобразява, се задава чрез свойството **Image**. Свойството **SizeMode** задава дали картинката да се разшири/намали или центрира при изобразяването в контролата.

Картинките, използвани в контролата **PictureBox**, се запазват като ресурси. Те се записват в XML формат в **.resx** файла на съответната форма и при компилация се запазват като ресурси в асемблито на приложението.

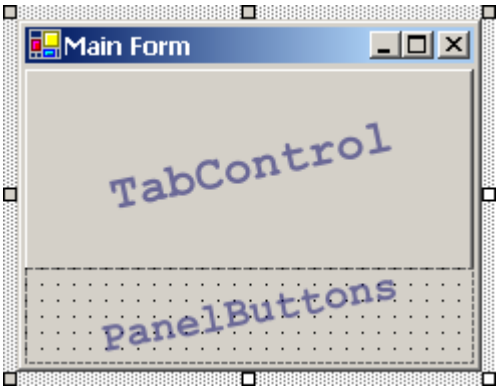
Работа с някои Windows Forms контроли – пример

В настоящия пример ще демонстрираме използването на някои от Windows Forms контролите, които разгледахме: **TabControl**, **TabPage**, **Panel**, **RadioButton**, **GroupBox**, **CheckedListBox**. За целта ще създадем малко приложение, което събира информация от потребителя и след това я показва в диалогов прозорец.

Ето стъпките за изграждане на нашето приложение:

1. Стартираме VS.NET и създаваме нов Windows Forms проект.
2. Задаваме на главната форма име **MainForm** и заглавие **"Main Form"**. Променяме и името на файла от **Form1.cs** на **MainForm.cs**.

3. Вземаме от ToolBox на VS.NET един **TabControl**, поставяме го в горната част на главната форма и му задаваме име **TabControl**. Под него поставяме един **Panel** и му задаваме име **PanelButtons**. На свойствата **Dock** на поставените контроли задаваме съответно стойности **Fill** (за таб контролата) и **Bottom** (за панела). По този начин, при оразмеряване (Resize) на формата, панелът ще си остава отдолу, като се разширява/намалява само странично, а поставеният **TabControl** ще заема цялото останало пространство. Ето как изглежда формата в този момент:



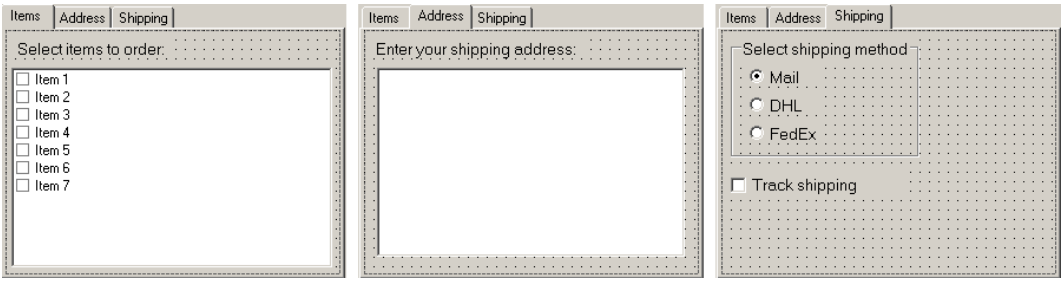
4. В **TabControl** контролата добавяме 3 страници (**TabPage** контроли)– първата за избиране на списък с продукти, втората за въвеждане на адрес и третата за избор на начин на доставка за поръчката. Подходящи имена за тези контроли са съответно **TabPageItems**, **TabPageAddress** и **TabPageShipping**. Добавянето на **TabPage** става, като щракнем с десния бутон на мишката върху **TabControl** контролата и от появилото се меню изберем **Add Tab**.

5. В страницата за избор на продукти добавяме една **CheckedListBox** контрола с име **CheckedListBoxItems** и я закотвяме за четирите страни на **TabPage** контролата чрез свойството **Anchor** от прозореца **Properties** на редактора. По този начин контролата ще се преоразмерява заедно с формата. Задаваме списък от продукти чрез свойството **Items** и добавяме над контролата един **Label** с текст "**Select items to order:**" и име **LabelSelectItem**.

6. В страницата за въвеждане на адрес добавяме една **TextBox** контрола с име **TextBoxAddress**, закотвяме я към четирите страни на страницата чрез свойството **Anchor**, задаваме на свойството **Multiline** стойност **true**, а на свойството **Text** задаваме празен низ. Добавяме над контролата един **Label** с текст "**Enter your shipping address:**" и име **LabelEnterAddress**.

7. В страницата за избор на начина на доставка добавяме една **GroupBox** контрола с име **GroupBoxShippingMethod** и текст **Select shipping method**. В нея добавяме три **RadioButton** контроли с имена

RadioButtonMail, **RadioButtonDHL** и **RadioButtonFedEx** и текст съответно "**Mail**", "**DHL**" и "**FedEx**". Тази комбинация от контроли ни осигурява алтернативен избор на един от възможните доставчици. Най-отдолу в тази страница добавяме и една **CheckBox** контрола с име **CheckBoxTrackShipping** и текст "**Track shipping**". Ето как изглеждат нашите табове в този момент:



8. В панела за бутоните добавяме два бутона с имена **ButtonOK** и **ButtonCancel** и текст съответно "**OK**" и "**Cancel**". Чрез двукратно щракване върху бутона **ButtonCancel** добавяме обработчик за събитието му **Click**, в който добавяме код за затваряне на формата:

```
private void ButtonCancel_Click(object sender,
   System.EventArgs e)
{
   Close();
}
```

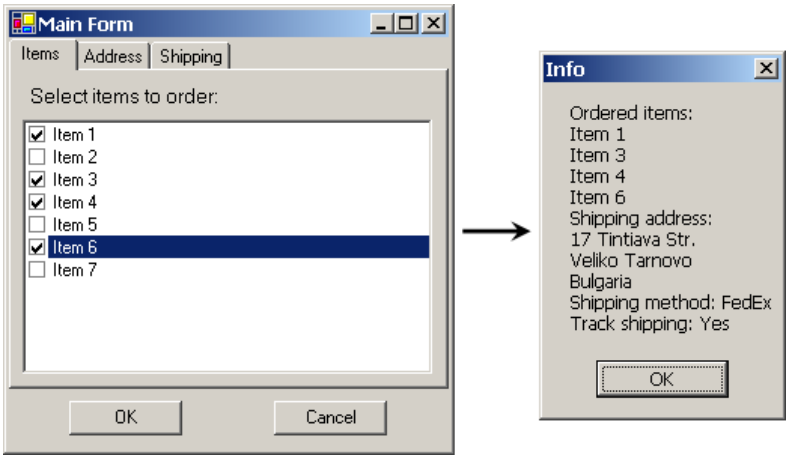
9. Остана само да дефинираме обработчик на събитието **Click** на бутона **ButtonOK**:

```
private void ButtonOK_Click(object sender, System.EventArgs e)
{
   StringBuilder order = new StringBuilder("Ordered items:\n");
   foreach (object item in CheckedListBoxItems.CheckedItems)
   {
      order.Append(item.ToString());
      order.Append("\n");
   }

   order.Append("Shipping address:\n");
   order.Append(TextBoxAddress.Text);

   order.Append("\nShipping method: ");
   if (RadioButtonMail.Checked)
   {
      order.Append("Mail");
   }
   if (RadioButtonDHL.Checked)
   {
```

```
      order.Append("DHL");
    }
    if (RadioButtonFedEx.Checked)
    {
      order.Append("FedEx");
    }

    order.Append("\nTrack shipping: ");
    if (CheckBoxTrackShipping.Checked)
    {
      order.Append("Yes");
    }
    else
    {
      order.Append("No");
    }

    MessageBox.Show(order.ToString(), "Info");

    Close();
}
```

При натискане на бутона извличаме стойностите, въведени във всички контроли, от всички страници, и ги записваме в символен низ. След това ги извеждаме на екрана в стандартна диалогова кутия. За да направим това, първо създаваме един **`StringBuilder`** обект, в който ще ги добавяме. След това добавяме стойностите на всички избрани елементи от **`CheckedListBoxItems`** контролата, като след всеки от тях добавяме нов ред. Добавяме адреса за доставка, после проверяваме кой **`RadioButton`** е избран и добавяме съответния метод за доставка към **`StringBuilder`** обекта. Накрая проверяваме състоянието на **`CheckBox`** контролата от страницата за начин на доставка и извеждаме извлечената от контролите информация в стандартна диалогова кутия.

10. Нашето приложение е готово и можем да го стартираме и тестваме:



Менюта

Менютата са важно средство, чрез което потребителят по удобен начин указва започването на дадена операция. Те са на практика стандарт при приложенията с графичен потребителски интерфейс. В Windows Forms за работа с менюта се използват класовете **`MainMenu`**, **`ContextMenu`** и **`MenuItem`**.

MainMenu

`MainMenu` компонентата представлява стандартно падащо меню. Тя съдържа в себе си списък от **`MenuItem`** елементи, които представят отделните възможности за избор (команди) от менюто.

ContextMenu

`ContextMenu` компонентата представлява контекстно меню (popup меню), което се появява, когато потребителят щракне с десния бутон на мишката върху контрола или някъде във формата. Тя съдържа списък от **`MenuItem`** елементи, представляващи отделните команди от менюто.

MenuItem

`MenuItem` елементите представляват отделните възможности за избор, показвани в **`MainMenu`** или **`ContextMenu`**. Всеки **`MenuItem`** елемент може да бъде команда в приложението или родителско меню за други елементи, (менютата могат да се влагат). По-важни събития и свойства на класа **`MenuItem`** са:

- **`Text`** – задава заглавието на елемента, например **`"&New"`**, **`"&Open…"`** или **`"-"`**. Символът **`"&"`** задава горещ клавиш за бързо избиране на съответния елемент. Поставя се преди дадена буква от текста на елемента. Елемент от менюто с текст **`"-"`** задава разделител.
- **`ShortCut`** – кратък клавиш, асоцииран с този елемент. Например може да се укаже, че **`[Ctrl+S]`** съответства на елемента от менюто File | Open. В такъв случай указаната клавишна комбинация, независимо кога е натисната, активира този елемент от менюто, стига това да се е случило при активна форма.
- **`Click`** – събитие, което се активира при избиране на елемента от менюто.

Ленти с инструменти

Лентите с инструменти са често използвани при приложенията с графичен интерфейс. Нека разгледаме стандартните средства на Windows Forms за работата с тях.

ToolBar

ToolBar контролата представлява лента с инструменти (с бутони). По-важни нейни свойства и събития са:

- **Buttons** – съдържа списък от бутоните (**ToolBarButton** елементите), асоциирани с контролата.
- **ImageList** – задава картинките за бутоните от лентата.
- **ButtonClick** – събитие, активиращо се при натискане на бутон от лентата. Като параметър то приема **ToolBarButtonClickEventArgs** с информация кой бутон е бил натиснат.

Не е ясно по каква причина, но проектантите на библиотеката Windows Forms са направили работата с ленти с инструменти доста неудобна. Вместо всеки бутон от лентата да си има собствено събитие **Click**, има едно общо събитие за цялата лента с инструменти, което се активира при натискане на някой от бутоните. Другият проблем е, че вместо всеки бутон да си има свойство **Picture**, картинките трябва да се слагат в **ImageList** компонента и да се ползват по индекс.

ToolBarButton

ToolBarButton компонентата представлява бутон от лентата с инструменти. За всеки бутон от лентата може да се задава картинка, която да се изобразява върху него, текстът за бутона, както и някои други свойства като **ToolTipText**, който да се показва при задържане на показалеца на мишката върху бутона.

Задаването на изображение на **ToolBarButton** става, като асоцииране **ImageList** с лентата с инструменти, в която се намира бутонът, и след това зададем на свойството **ImageIndex** на бутона стойност с индекса на изображението.

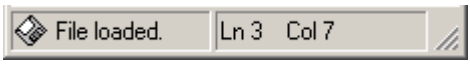
ImageList

ImageList компонентата съдържа списък с картинки. Тя често се използва от други компоненти като **ListView**, **TreeView** или **ToolBar**.

Статус ленти

Статус лентите са още една от типичните контроли в приложенията с графичен интерфейс. Те се използват за визуализация на информация, свързана със състоянието на приложението. Например в текстовите редактори много често в статус лентата се показва номерът на текущия ред и на колона.

StatusBar



StatusBar контролата представлява лента за състоянието. Тя обикновено се състои от **StatusBarPanel** обекти, показващи текст или икони. Тя може да съдържа и потребителски дефинирани панели, показващи информация за състоянието на приложението. По-важни свойства на контролата са:

- **Panels** – съдържа секциите, на които е разделена лентата. На фигурата по-горе статус лентата е разделена на 2 секции. Статус лентата може и да няма отделни секции, а да е едно цяло.
- **ShowPanels** – включва/изключва показването на панелите. Ако секциите са изключени, статус лентата е едно цяло.

StatusBarPanel

StatusBarPanel компонентата представлява секция в лентата за състоянието. Тя може да съдържа текст и/или икона. Чрез свойството **Text** се задава текстът, който се показва в панела, а чрез свойството **Icon** се задава икона.

Диалог за избор на файл

При графичните приложения често се налага да се избира файл от локалната файлова система, например, когато трябва да бъде зареден или записан документ във файл. За тази цел Windows Forms предоставя стандартни диалози, които могат да се настройват по гъвкав начин.

OpenFileDialog

OpenFileDialog представлява диалог за избор на файл (при отваряне). Този клас ни позволява да проверим дали файл съществува и да го отворим. По-важни свойства на диалога са:

- **Title** – задава заглавие на диалога.
- **InitialDirectory** – задава началната директория, от която започва изборът на файл. Ако не бъде зададена изрично, се използва последната директория, от която потребителят е избирал файл по време на работа с текущото приложение.

- **Filter** – задава възможните файлови разширения, измежду които потребителят може да избира (например ***.txt**, ***.doc**, ...).
- **FilterIndex** – задава активния филтър.
- **MultiSelect** – указва дали в диалога могат да бъдат избирани много файлове едновременно или само един.
- **FileName**, **FileNames** – съдържа избраните файлове.

SaveFileDialog

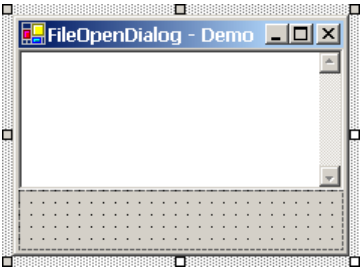
SaveFileDialog представлява диалог за избор на файл (при записване). Този клас ни позволява да презапишем съществуващ или да създадем нов файл. Работата с него е аналогична на работата с **OpenFileDialog**.

Работа с файлов диалог – пример

Настоящият пример илюстрира работата с файловия диалог на Windows Forms (компонентата **OpenFileDialog**). За целта ще създадем приложение, което позволява на потребителя да избере текстов файл с помощта на **OpenFileDialog**, чете съдържанието му и го показва в текстова контрола.

Ето стъпките за изграждане на нашето приложение:

1. Стартираме VS.NET и създаваме нов Windows Forms проект.
2. Задаваме на главната форма име **MainForm** и заглавие **"FileOpenDialog - Demo"**. Променяме името на файла от **Form1.cs** на **MainForm.cs**.
3. Вземаме от ToolBox на VS.NET един **TextBox**, поставяме го в горната част на главната форма и му задаваме име **textBox**. Задаваме на свойството му **Multiline** стойност **true** и на свойството му **ScrollBars** стойност **Vertical**. Така си осигуряваме многоредово текстово поле с възможност за скролиране. Под него поставяме един **Panel** и му задаваме име **PanelBottom**. На свойствата **Dock** на поставените контроли задаваме съответно стойности **Fill** и **Bottom**. По този начин, при оразмеряване (Resize) на формата, панелът ще си остава отдолу, като се разширява/намалява само странично, а поставеният **TextBox** ще заема цялото останало пространство. Ето как изглежда формата в този момент:



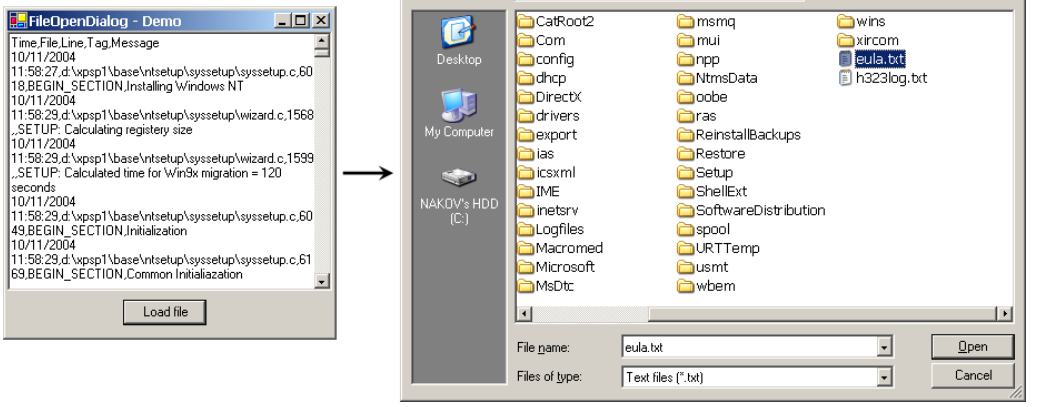
4. Поставяме във формата един **OpenFileDialog** с име **openFileDialog**. Задаваме на свойството **Filter** стойност "**Text files (*.txt)|*.txt|Log files (*.log)|*.log**". Този филтър указва търсене само на текстови (**.txt**) и log (**.log**) файлове. На свойството **Title** задаваме стойност "**Choose text file**".

5. В панела добавяме един бутон с име **ButtonLoadFile** и текст "**Load file**". Чрез двукратно щракване върху бутона добавяме обработчик за събитието му **Click**:

```
private void ButtonLoadFile_Click(object sender,
   System.EventArgs e)
{
   if (openFileDialog.ShowDialog() == DialogResult.OK)
   {
      string fileName = openFileDialog.FileName;
      using (StreamReader reader = File.OpenText(fileName))
      {
         string fileContents = reader.ReadToEnd();
         textBox.Text = fileContents;
      }
   }
}
```

При натискане на бутона показваме диалог за избор на файл и ако потребителят избере файл и натисне бутона **[OK]**, отваряме файла, четем съдържанието му и го показваме в текстовото поле.

6. Нашето приложение е готово и можем да го стартираме и тестваме:



MDI приложения

MDI (Multiple Document Interface) приложенията поддържат работа с няколко документа едновременно, като всеки документ се показва в свой собствен прозорец, разположен във вътрешността на главния прозорец.

MDI контейнери (MDI parents)

MDI контейнерите са форми, които съдържат други форми. За да укажем, че една форма е MDI контейнер, задаваме на нейното свойство **IsMdiContainer** стойност **true**. Тези форми обикновено имат меню **Window** за смяна на активната форма (на свойството му **MdiList** е зададена стойност **true**).

MDI форми (MDI children)

MDI формите се съдържат в контейнер-формата. За да укажем, че една форма е MDI форма, задаваме на свойство **MdiParent=<контейнер>**, където **контейнер** е MDI контейнер.

Създаване на многодокументов текстов редактор – пример

С настоящия пример ще демонстрираме изграждане на многодокументов текстов редактор със средствата на Windows Forms и Visual Studio .NET. Редакторът трябва да може да създава, редактира, зарежда и записва текстови документи (файлове) и да позволява работа едновременно с много документи в отделни MDI прозорци.

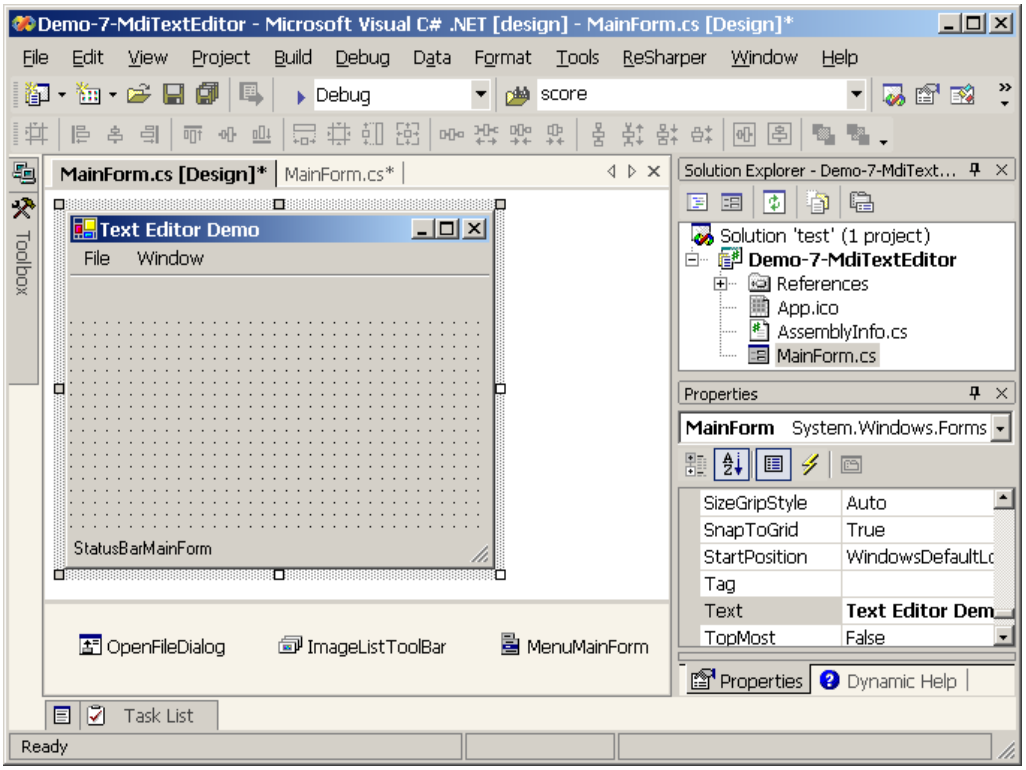
Чрез примерния текстов редактор ще демонстрираме употребата на някои от Windows Forms контролите, които разгледахме: менюта (**MainMenu**, **MenuItem**), ленти с инструменти (**ToolBar**, **ImageList**, **ToolBarButton**) и статус ленти (**StatusBar**, **StatusBarPanel**). Ще покажем как се създават приложения, работещи в MDI режим. Ще демонстрираме работата с диалога за избор на файл.

Ето стъпките за изграждането на нашия текстов редактор:

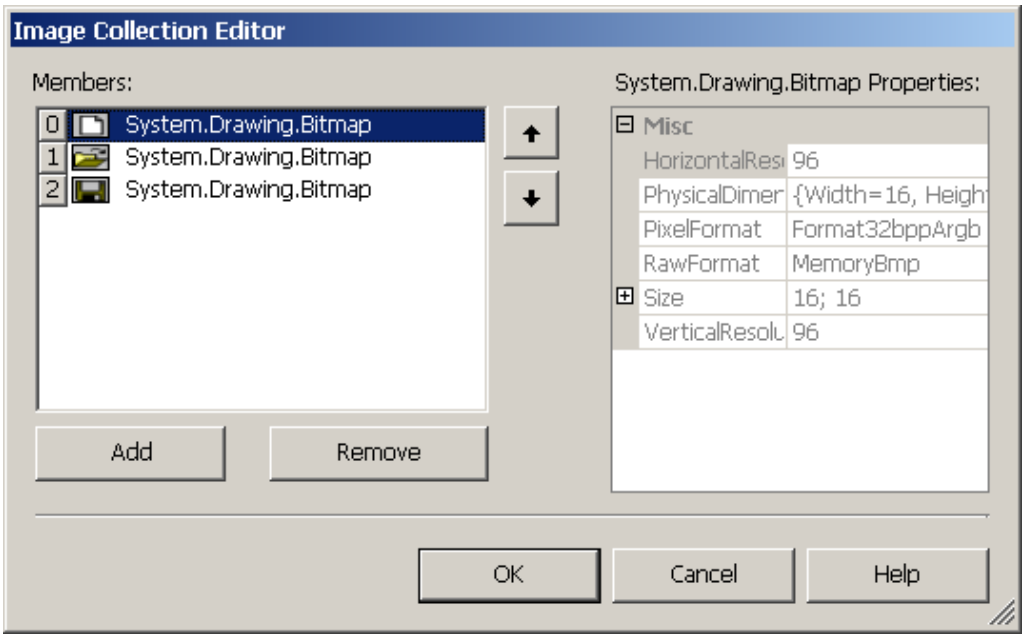
1. Стартираме VS.NET и създаваме нов Windows Forms проект.
2. Взимаме от ToolBox на VS.NET едно **MainMenu**, един **ToolBar**, един **ImageList**, един **StatusBar** и един **OpenFileDialog** и ги поставяме в главната форма. Задаваме подходящи имена на поставените компоненти. Препоръчително е името на една контрола да съдържа нейното предназначение и тип (или префикс, указващ типа). В нашия случай подходящи имена са: **MenuMainForm**, **ToolBarMainForm**, **ImageListToolBar**, **StatusBarMainForm** и **OpenFileDialog**.
3. Задаваме за филтър на **OpenFileDialog** контролата стойността "**Text files (*.txt)|*.txt**". Този филтър указва търсене само на текстови файлове (**.txt**).
4. Дефинираме **File** и **Window** менюта в главното меню (засега ще ги оставим празни, без елементи в тях).

5. Задаваме на главната форма име **`MainForm`** и заглавие **`"Text Editor Demo"`**. Променяме и името на файла от **`Form1.cs`** на **`MainForm.cs`**. На картинката по-долу е показано как изглежда разработваното приложение в този момент.

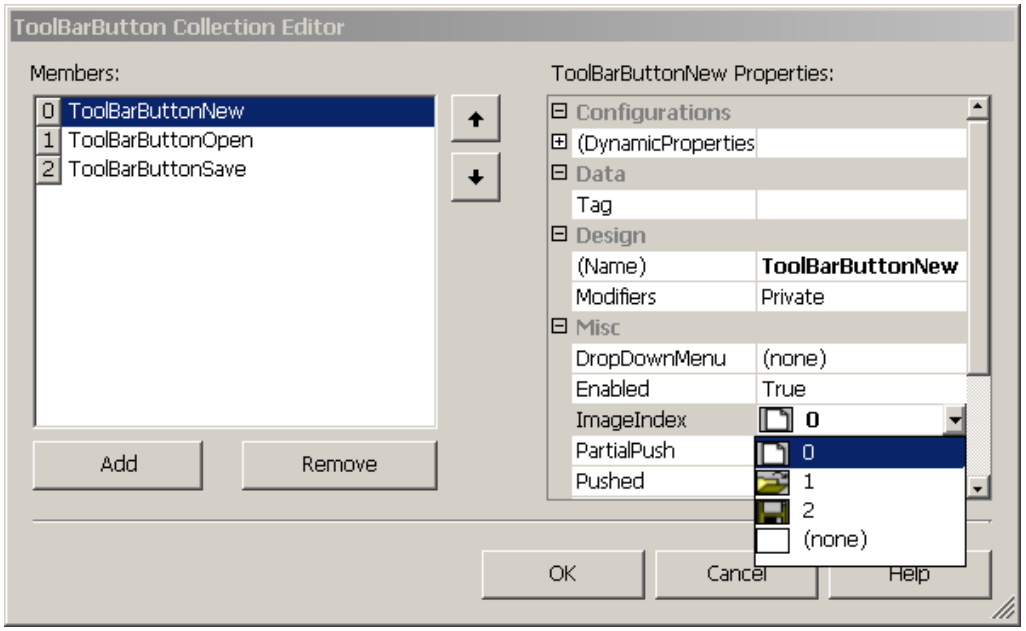
6. Преди да дефинираме бутоните в лентата с инструменти, трябва да заредим подходящи иконки за тях в **`ImageList`** контролата. Трябват ни иконка за нов файл, за отваряне на файл и за запис на файл. Можем да използваме стандартните иконки, идващи с VS.NET. Те се намират в директория: **`C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\Graphics\bitmaps\OffCtlBr\Small\Color`** (при стандартна инсталация на Visual Studio .NET).



7. От редактора за свойствата на компонентите избираме свойството **`Images`** на **`ImageList`** контролата. Появява се Image Collection Editor, от който можем да добавим иконки в списъка. Добавяме подходящи иконки за нов файл, за отваряне на файл и за запис на файл:

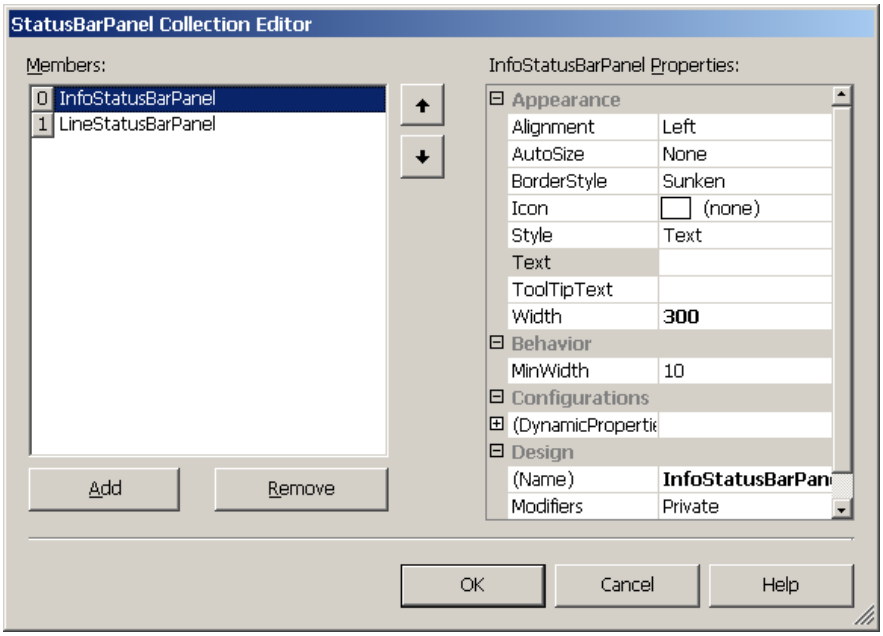


8. За да дефинираме бутоните в лентата с инструменти, първо свързваме **`ImageList`** свойството на **`ToolBar`** контролата с **`ImageList`** компонентата, която заредихме с иконки в предната стъпка. След това използваме свойството **`Buttons`** на поставената във формата **`ToolBar`** контрола, за да дефинираме бутоните. За редакция на това свойство се използва ToolBarButton Collection Editor, който се появява при опит за редактиране на свойството **`Buttons`**:



Добавяме три бутона (за нов файл, за отваряне на файл и за запис на файл) и задаваме за всеки от тях подходящо име и **`ImageIndex`**, който го свързва с неговата иконка от **`ImageList`** контролата. В този момент в лентата с инструменти се появяват трите бутона с иконки върху тях:

9. Статус лентата ще разделим на две части. В лявата част ще показваме информация за извършените от приложението действия, а в дясната – номера на реда в текущия файл. За целта задаваме на статус лентата **`ShowPanels=true`** и добавяме в нея два панела (чрез свойството **`Panels`**). Задаваме им имената **`StatusBarPanelInfo`** и **`StatusBarPanelLine`** и им настройваме размерите:

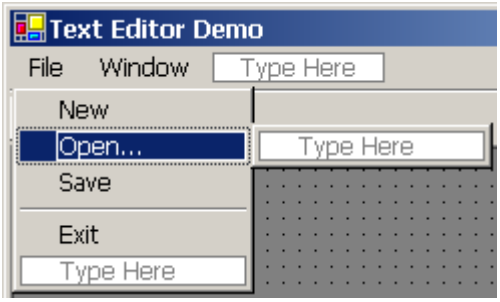


Статус лентата добива следния вид:

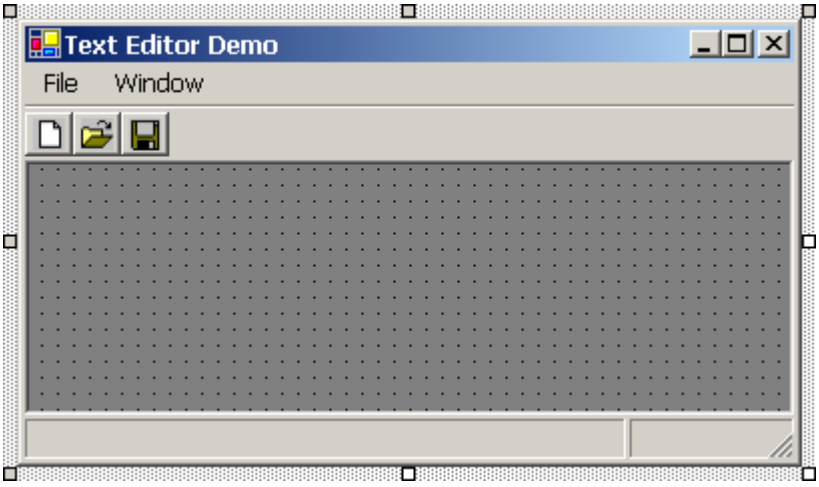
10. За да направим главната форма MDI форма, й задаваме **`IsMdiContainer=true`**.

11. Създаваме елементите на главното меню **`File`**: **`New`**, **`Open`**, **`Save`** и **`Exit`**. За да създадем разделител преди елемента **`Exit`**, задаваме на съответната **`MenuItem`** контрола **`Text="-"`**. За **`Window`** менюто задаваме **`MdiList=true`**, за да показва списък от MDI прозорците в главната форма. За елементите на менюто избираме подходящи имена (например **`MenuItemFileNew`**, **`MenuItemFileOpen`**, ...). Задаваме и подходящи бързи клавиши (shortcuts) за често използваните команди чрез свойството **`Shortcut`** на **`MenuItem`** контролата –

[Ctrl+N] за **File** | **New**, **[Ctrl+O]** за **File** | **Open**, **[Ctrl+S]** за **File** | **Save** и **[Alt-F4]** за **File** | **Exit**. Ето как изглежда главното меню в този момент:



Цялата форма на приложението добива следния вид:



12. Вече имаме главната форма. Остава да добавим формата за редактиране на файловете и да реализираме логиката на приложението. Започваме от дефиниране на събитията за елементите от менюто. С двойно щракване върху елемент от менюто VS.NET ни дава възможност да напишем кода за обработка на събитието му **Click**:

```
private void MenuItemFileNew_Click(object sender,
   System.EventArgs e)
{
   CreateNewFile();
}

private void MenuItemFileOpen_Click(object sender,
   System.EventArgs e)
{
   OpenExistingFile();
}
```

```
private void MenuItemFileSave_Click(object sender,
   System.EventArgs e)
{
   SaveCurrentFile();
}

private void MenuItemFileExit_Click(object sender,
   System.EventArgs e)
{
   Close();
}
```

Методите **CreateNewFile()**, **OpenExistingFile()** и **SaveCurrentFile()** ще разгледаме след малко.

13. Дефинираме и обработчик на събитието натискане на бутон от лентата с инструменти:

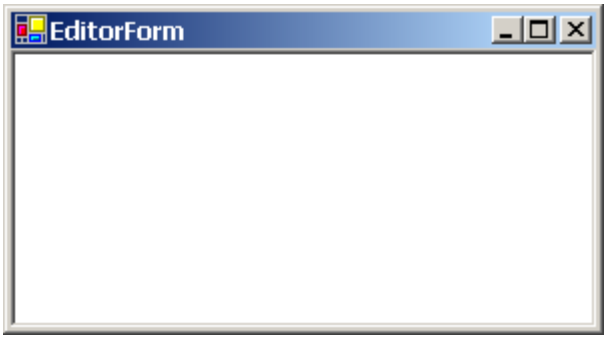
```
private void ToolBarMainForm_ButtonClick(object sender,
   System.Windows.Forms.ToolBarButtonClickEventArgs e)
{
   if (e.Button == ToolBarButtonNew)
   {
      CreateNewFile();
   }
   else if (e.Button == ToolBarButtonOpen)
   {
      OpenExistingFile();
   }
   else if (e.Button == ToolBarButtonSave)
   {
      SaveCurrentFile();
   }
}
```

Понеже контролата **ToolBar** не предоставя отделни събития за всеки от бутоните си, трябва да се прихване събитието й **ButtonClick** и да се проверява за кой от бутоните се отнася то (чрез свойството **Button** на **ToolBarButtonClickEventArgs** параметъра).

14. Остава да дефинираме формата за редакция на документ и да реализираме логиката за създаване, редактиране, зареждане и записване на документи. Създаваме нова форма (**File** | **Add New Item** … | **Windows Form**). Сменяме й името на **EditorForm**, а името на нейния файл – на **EditorForm.cs**. Тази форма ще служи за редакция на документите. Тя ще се използва като подчинена MDI форма.

15. Добавяме **RichTextBox** контрола в новата форма. Тя ще служи за редакция на текста на документите. Използваме **RichTextBox** вместо **TextBox**, защото **RichTextBox** позволява работа с по-големи

документи и осигурява по-голяма гъвкавост. Задаваме **Dock=Fill** за **RichTextBox** контролата и й сменяме името на **EditorRichTextBox**. Ето как изглежда формата след всички тези действия:



16. Дефинираме в новата форма поле **mFileName**, което ще съхранява името на текущия отворен файл или **null**, ако текущият файл няма име (например ако е нов файл):

```
private string mFileName = null;
```

17. Поставяме в новата форма един **SaveFileDialog**. Ще го ползваме при запис на файла, който е зареден в **RichTextBox** контролата. Задаваме му филтър "**Text files (*.txt)|*.txt**".

18. Дефинираме няколко метода, които реализират логиката по отваряне на нов документ, зареждане на файл и записване на файл, както и помощен метод за обновяване на статус лентата:

```
public void CreateNewFile()
{
   SetStatusBarInfo("Created new file.");
   mFileName = null;
   this.Text = "Untitled";
}

public void LoadFile(string aFileName)
{
   mFileName = aFileName;
   this.Text = Path.GetFileName(aFileName);
   using (StreamReader reader = File.OpenText(aFileName))
   {
      string fileContents = reader.ReadToEnd();
      RichTextBoxEditor.Text = fileContents;
   }
}

public void Save()
{
   if (mFileName == null)
```

```
   {
      if (SaveFileDialog.ShowDialog() != DialogResult.OK)
      {
         return;
      }
      mFileName = SaveFileDialog.FileName;
      this.Text = Path.GetFileName(mFileName);
   }

   using (StreamWriter writer = new StreamWriter(mFileName))
   {
      writer.Write(RichTextBoxEditor.Text);
   }

   SetStatusBarInfo("Saved file: " + mFileName);
}

public void SetStatusBarInfo(string aText)
{
   MainForm mainForm = (MainForm) this.MdiParent;
   mainForm.SetInfoStatusBar(aText);
}
```

Създаването на нов документ задава заглавие "**Untitled**" на формата и установява в **null** името на файла, свързан с нея. Зареждането на файл става с текстов поток. При зареждане формата запомня пълното име на файла, а за заглавие на формата се задава името на файла без пътя. При запис, ако документът не е свързан с файл, се използва файловият диалог за избор на име на файл, в който да се запише. Ако документът е свързан с файл, той просто се записва. Записът става с текстов поток.

19. Дефинираме няколко обработчика на събития и няколко помощни метода с цел визуализация на номера на реда в текущия файл:

```
private void EditorForm_Activated(object sender,
   System.EventArgs e)
{
   ShowLineNumber();
}

private void RichTextBoxEditor_SelectionChanged(object sender,
   System.EventArgs e)
{
   ShowLineNumber();
}

public void SetStatusBarLine(string aText)
{
   MainForm mainForm = (MainForm) this.MdiParent;
```

```
   mainForm.SetLineStatusBar(aText);
}

public void ShowLineNumber()
{
   int currentPos = EditorRichTextBox.SelectionStart;
   int line = RichTextBoxEditor.GetLineFromCharIndex(currentPos);
   SetStatusBarLine("Line: " + line);
}
```

При активиране на формата и при промяна на позицията на курсора приложението изчислява номера на текущия ред в текущия документ и го показва в десния панел на лентата за състоянието. Достъпът до лентата на състоянието става през родителската MDI форма (това е главната форма на приложението).

20. Дефинираме и обработчик на събитието "затваряне на формата", в който извеждаме информация в статус лентата какво се е случило:

```
private void EditorForm_Closed(object sender,
   System.EventArgs e)
{
   if (mFileName != null)
   {
      SetStatusBarInfo("Closed file: " + mFileName);
   }
   else
   {
      SetStatusBarInfo("Closed file.");
   }
   SetStatusBarLine("");
}
```

С това формата за редактиране на файлове е готова. Остава само да довършим главната форма и приложението ще е готово.

21. В главната форма пропуснахме да дефинираме методите за отваряне на нов файл, за зареждане на съществуващ файл и за затваряне на файл. Ето как можем да ги реализираме:

```
private void CreateNewFile()
{
   EditorForm editorForm = new EditorForm();
   editorForm.MdiParent = this;
   editorForm.CreateNewFile();
   editorForm.Show();
}

private void OpenExistingFile()
{
```

```
   if (OpenFileDialog.ShowDialog() != DialogResult.OK)
   {
      return;
   }

   string fileName = OpenFileDialog.FileName;

   EditorForm editorForm = new EditorForm();
   try
   {
      editorForm.LoadFile(fileName);
      editorForm.MdiParent = this;
      editorForm.Show();
      SetInfoStatusBar("Loaded file: " + fileName);
   }
   catch (IOException)
   {
      editorForm.Dispose();
      MessageBox.Show("Can not load file: " + fileName, "Error");
   }
}

private void SaveCurrentFile()
{
   EditorForm activeEditorForm =
      (EditorForm) this.ActiveMdiChild;
   if (activeEditorForm != null)
   {
      activeEditorForm.Save();
   }
}
```

При създаване и зареждане на файл се създава инстанция на формата за редакция на документи **`EditorForm`** и в нея съответно се създава нов документ или се зарежда избрания чрез **`OpenFileDialog`** файл, след което тази форма се показва като MDI подчинена в главната.

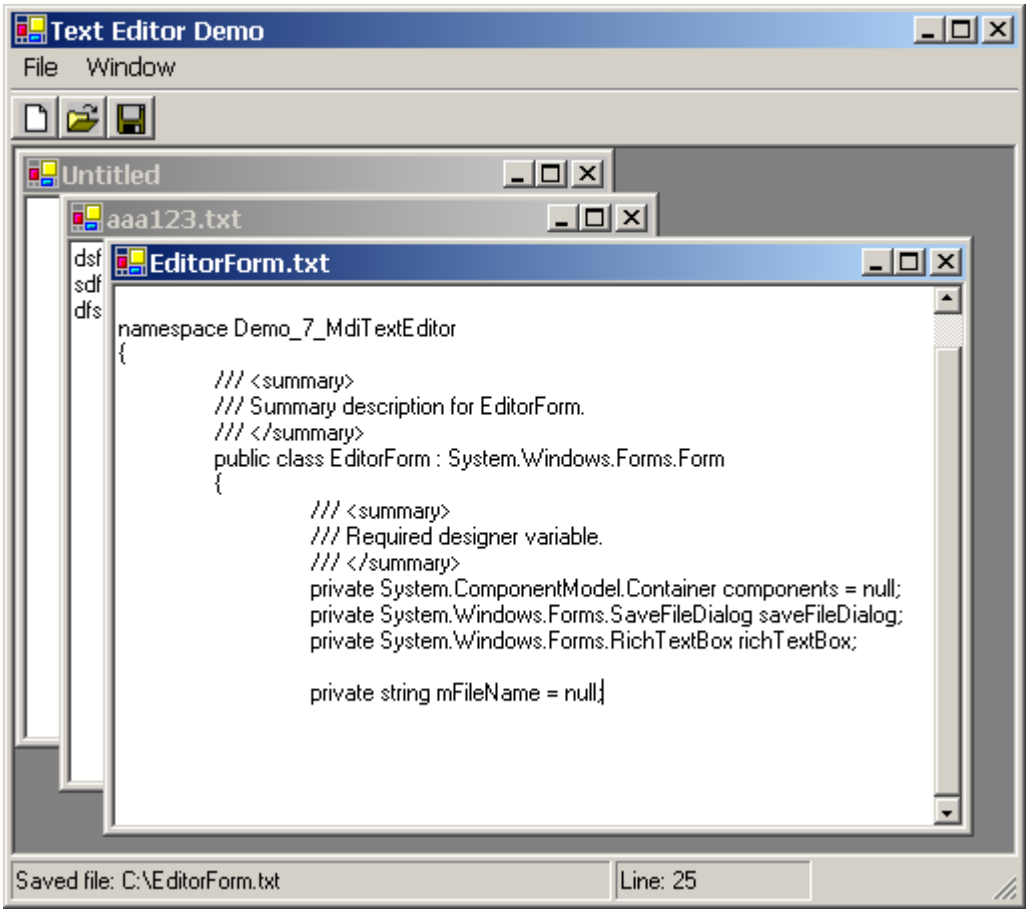
При записване на текущия документ първо се извлича текущата активна форма (ако има такава) и след това й се извиква методът **`Save()`** за записване на отворения в нея документ.

22. Остана само да дефинираме още няколко обработчика на събития за главната форма и няколко помощни метода, които използваме:

```
private void MainForm_Load(object sender, System.EventArgs e)
{
   SetInfoStatusBar("Application started.");
}

public void SetInfoStatusBar(string aText)
```

```
{
   StatusBarPanelInfo.Text = aText;
}

public void SetLineStatusBar(string aText)
{
   StatusBarPanelLine.Text = aText;
}

static void Main()
{
   Application.Run(new MainForm());
}
```

23. Приложението вече е готово и можем да го стартираме и тестваме. Ето как изглежда нашият текстов редактор в действие:



Валидация на данни

Валидацията на данни е необходима, когато в дадена контрола трябва да се допуска въвеждане само на коректни данни, например цяло число, валидна дата и др. В Windows Forms има стандартни средства за валидация:

- **Validating** – събитие за валидация на данните в класа **Control**. На събитието се подава параметър от тип **CancelEventArgs**. Ако на свойството **Cancel** на този обект се зададе стойност **true**, то на потребителя не се разрешава да напусне контролата.
- **ErrorProvider** – отбелязва графично контроли с невалидни данни. До контролата с невалидни данни се появява икона, а когато показалецът на мишката застане над иконата, се появява текст с описание на грешката.

Нека разгледаме следващия фрагмент код, илюстриращ валидацията на данни:

```
private TextBox TextBox1;
private ErrorProvider errorProvider;
...

private void TextBox1_Validating(object sender,
   System.ComponentModel.CancelEventArgs e)
{
   try
   {
      Int32.Parse(TextBox1.Text);
      errorProvider.SetError(TextBox1, "");
   }
   catch (FormatException)
   {
      errorProvider.SetError(
         TextBox1, "Integer number expected!");
      e.Cancel = true;
   }
}
```

Имаме една **TextBox** контрола, чиито данни ще валидираме, и един **ErrorProvider** обект, който ще използваме, за да отбелязваме, че контролата съдържа невалидни данни.

В обработчика на събитието **Validating** на контролата се опитваме да конвертираме текста, съдържащ се в нея, в цяло число. Ако конвертирането пропадне, това означава, че потребителят не е въвел коректни данни. В този случай подаваме на метода **SetError(…)**, на **ErrorProvider** обекта, нашата контрола и символен низ с описание на грешката. Това описание ще се появи при задържане на мишката над иконата за грешка. Освен това задаваме на свойството **Cancel** на подадения **CancelEventArgs**

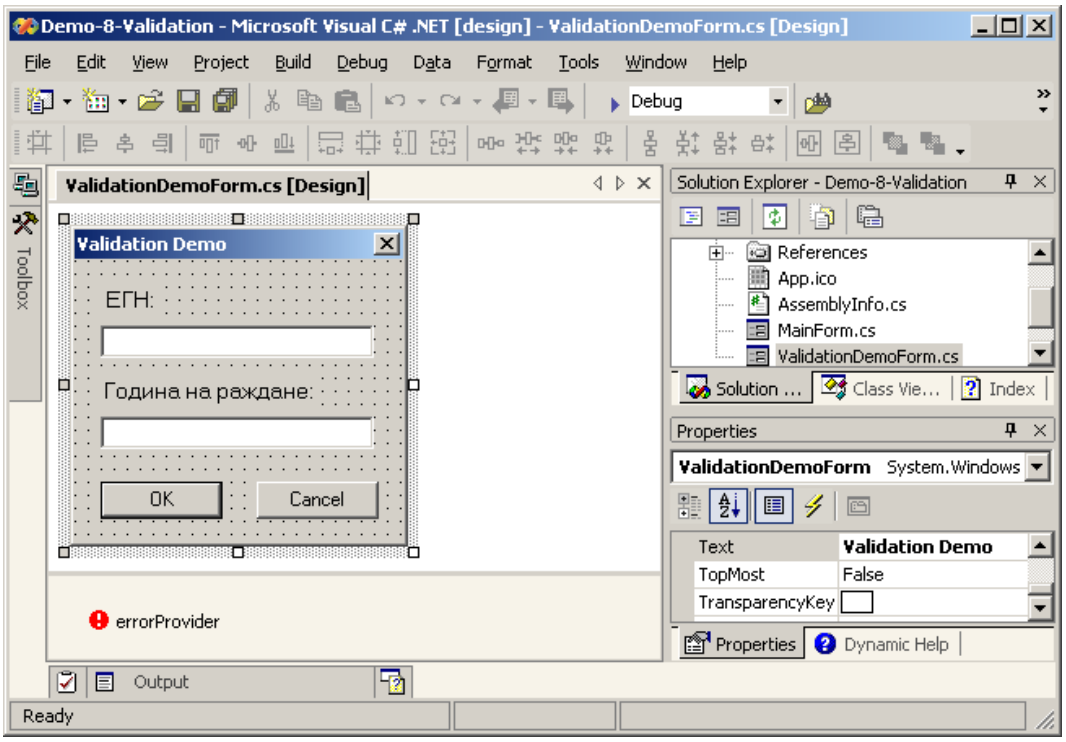
обект стойност **true**. Това няма да позволи на потребителя да напусне контролата. Ако конвертирането успее, то потребителят е въвел коректни данни. В този случай отново извикваме метода **SetError(…)**, но този път му подаваме като втори параметър празен низ, което предизвиква скриване на иконата, ако тя е била показана.

Валидация на данни – пример

Настоящият пример е малко по-сложен и илюстрира по-пълно средствата за валидация на данни в Windows Forms – събитието **Validating** и контролата **ErrorProvider**. Ще създадем просто приложение, състоящо се от две форми – главна форма и форма за въвеждане на ЕГН и година на раждане. Главната форма ще извиква формата за въвеждане на ЕГН и година на раждане и при успешно връщане от нея ще визуализира въведените данни. Във формата за въвеждане на ЕГН и година на раждане ще сигнализираме на потребителя, когато той въведе некоректни данни.

Ето стъпките за изграждане на нашето приложение:

1. Стартираме VS.NET и създаваме нов Windows Forms проект.
2. Задаваме на главната форма име **MainForm** и заглавие "**Main Form**". Променяме и името на файла от **Form1.cs** на **MainForm.cs**. Създаваме и формата за въвеждане на ЕГН и година на раждане (**File | Add New Item … | Windows Form**). Сменяме името й на **ValidationDemoForm**, а това на файла й на **ValidationDemoForm.cs**. Задаваме на свойствата **MinimizeBox** и **MaximizeBox** стойности **false**, а на свойството **FormBorderStyle** стойност **FixedDialog**.
3. В новосъздадената форма поставяме две текстови полета с имена **TextBoxEGN** и **TextBoxYear** за въвеждане на ЕГН и година на раждане. Над всяко от тях поставяме по един **Label** с текст, указващ предназначението на контролата. Поставяме и два бутона с имена **ButtonOK** и **ButtonCancel** за потвърждаване и отказване на формата. На свойството **DialogResult** на **ButtonCancel** задаваме стойност **Cancel**.
4. Поставяме във формата един компонент **ErrorProvider** с име **errorProvider**, който ще използваме за отбелязване на контролите с невалидни данни. Ето как изглежда на формата в този момент:



5. Добавяме обработчик на събитието **`Validating`** на **`TextBoxEGN`** контролата:

```
private void TextBoxEGN_Validating(object sender,
   System.ComponentModel.CancelEventArgs e)
{
   ValidateEGN();
}

private bool ValidateEGN()
{
   if (IsEgnValid(TextBoxEGN.Text))
   {
      errorProvider.SetError(TextBoxEGN, "");
      return true;
   }
   else
   {
      errorProvider.SetError(TextBoxEGN, "Невалидно ЕГН!");
      return false;
   }
}

private bool IsEgnValid(string aText)
{
   if (aText.Length != 10)
```

```
   {
      return false;
   }

   for (int i=0; i<aText.Length; i++)
   {
      if (! Char.IsDigit(aText[i]))
      {
         return false;
      }
   }

   return true;
}
```

В обработчика на събитието извикваме функцията **ValidateEGN()**. В нея, чрез функцията **IsEgnValid(…)**, проверяваме дали въведеното ЕГН е валидно. Ако е валидно, посредством **ErrorProvider** обекта, изтриваме маркера за грешка до полето за въвеждане на ЕГН и връщаме стойност **true**, в противен случай задаваме маркер за грешка на полето и връщаме стойност **false**. Във функцията **IsEgnValid(…)** проверяваме дали в полето за ЕГН са въведени десет символа и дали всеки от тях е цифра. Ако е така връщаме стойност **true**, в противен случай връщаме стойност **false**.

6. Добавяме обработчик на събитието **Validating** на **TextBoxYear** контролата:

```
private void TextBoxYear_Validating(object sender,
   System.ComponentModel.CancelEventArgs e)
{
   ValidateYear();
}

private bool ValidateYear()
{
   if (IsYearValid(TextBoxYear.Text))
   {
      errorProvider.SetError(TextBoxYear, "");
      return true;
   }
   else
   {
      errorProvider.SetError(TextBoxYear, "Невалидна година!");
      return false;
   }
}

private bool IsYearValid(string aText)
```

```
{
   string year = TextBoxYear.Text;
   if (year.Length != 4)
   {
      return false;
   }

   for (int i=0; i<aText.Length; i++)
   {
      if (! Char.IsDigit(aText[i]))
      {
         return false;
      }
   }

   return true;
}
```

В обработчика на събитието извикваме функцията **ValidateYear()**. В нея, чрез функцията **IsYearValid(…)**, проверяваме дали въведената година е валидна. Ако е валидна, посредством **errorProvider** обекта, изтриваме маркера за грешка до полето за въвеждане на година и връщаме стойност **true**, в противен случай задаваме маркер за грешка на полето и връщаме стойност **false**. Във функцията **IsYearValid(…)** проверяваме дали в полето за година са въведени четири символа и дали всеки от тях е цифра. Ако е така, връщаме стойност **true**, в противен случай връщаме стойност **false**.

7. Добавяме обработчик на събитието **Click** на бутона **ButtonOK**:

```
private void ButtonOK_Click(object sender, System.EventArgs e)
{
   if (ValidateForm())
   {
      DialogResult = DialogResult.OK;
   }
   else
   {
      MessageBox.Show(
         "Моля въведете валидни стойности във всички полета!",
         "Грешка", MessageBoxButtons.OK, MessageBoxIcon.Error);
   }
}

private bool ValidateForm()
{
   if (! ValidateYear())
   {
      return false;
```

```
    }

    if (! ValidateEGN())
    {
      return false;
    }

    string egn = TextBoxEGN.Text;
    string year = TextBoxYear.Text;
    if (egn.Substring(0,2) == year.Substring(2,2))
    {
      errorProvider.SetError(ButtonOK, "");
      return true;
    }
    else
    {
      errorProvider.SetError(ButtonOK,
        "Годината на раждане на съответства на ЕГН-то!");
      return false;
    }
}
```

При натискане на бутона проверяваме чрез функцията **ValidateForm()** дали данните, въведени във формата са валидни. Ако са валидни, задаваме на свойството **DialogResult** на формата стойност **DialogResult.OK**, с което връщаме управлението на извикващата форма. Ако данните са невалидни, показваме диалогова кутия с подходящо съобщение.

В метода **ValidateForm()** проверяваме дали въведените година и ЕГН са валидни чрез функциите **ValidateYear()** и **ValidateEGN()**. Ако проверката на някое от тези условия пропадне, връщаме стойност **false**. След това проверяваме дали първите две цифри на ЕГН-то съвпадат с последните две цифри на годината на раждане. Ако съвпадат, посредством **ErrorProvider** обекта, изтриваме маркера за грешка до бутона и връщаме стойност **true**. Ако цифрите се различават, задаваме маркер за грешка на бутона и връщаме стойност **false**.

8. Добавяме свойства, чрез които да извличаме въведените във формата ЕГН и година на раждане:

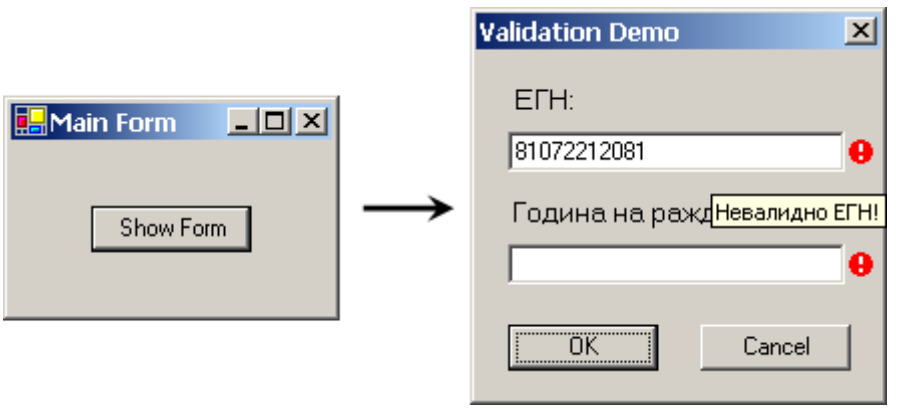
```
public string EGN
{
  get
  {
    return TextBoxEGN.Text;
  }
}
```

```
public string Year
{
   get
   {
      return TextBoxYear.Text;
   }
}
```

9. Добавяме в главната форма бутон с име **ButtonShow** и текст **Show Form**. В обработчика на събитието **Click** на този бутон ще извикваме формата за въвеждане на ЕГН и година на раждане в модален режим и при успешно връщане от нея ще визуализираме въведените данни:

```
private void ButtonShow_Click(object sender, System.EventArgs e)
{
   ValidationDemoForm validationDemoForm = new
      ValidationDemoForm();
   if (validationDemoForm.ShowDialog() == DialogResult.OK)
   {
      string s = String.Format("ЕГН: {0}\nГодина: {1}",
         validationDemoForm.EGN, validationDemoForm.Year);
      MessageBox.Show(s, "Резултат");
   }
   validationDemoForm.Dispose();
}
```

10. Приложението вече е готово и можем да го стартираме и тестваме:



Свързване на данни

Свързването на данни (data binding) осигурява автоматично прехвърляне на данни между контроли и източници на данни. Можем например да свържем масив, съдържащ имена на градове, с **ComboBox** контрола и имената от масива ще се показват в нея.

При добавянето на свързване указваме свойството на контролата, което ще свързваме с данните, източника на данните и път до списък или свойство на източника, към което ще се свържем. Този път може да е име на свойство, йерархия от имена, разделени с точки, или празен низ. Ако пътят е празен низ, ще се извика методът **`ToString()`** на обекта, използван като източник на данни.

Свързването на данни е еднопосочно – от контролата към източника на данни!

Промяна на дадено свързано свойство от дадена контрола променя данните в източника, към който то е свързано. Обратното не е вярно. При промяна на източника на данни свързаните към него контроли не си променят свойствата.

Ако сме променили данните в източника на данни и искаме да отразим промените в свързаните с него контроли, трябва първо да премахнем (изтрием) свързването и след това да го добавим отново.

Източници на данни

Като източник на данни можем да използваме всеки клас или компонент, който имплементира интерфейса **`IList`**. Такива са масивите и колекциите. За източници на данни можем да използваме и класове или компоненти, които имплементират интерфейса **`IBindingList`**. Този интерфейс поддържа нотификация за промяна на данните. **`IBindingList`** интерфейсът се имплементира от класа **`DataView`** (вж. темата за ADO.NET).

Контроли, поддържащи свързване на данни

Всички Windows Forms контроли поддържат свързване на данни (data binding). Можем да свържем което и да е свойство на контрола към източник на данни.

Видове свързване

В Windows Forms имаме два типа свързване на данни:

- Просто свързване (simple binding) – свързване на контрола с единичен обект или единичен (текущ) елемент от списък. Такова свързване използваме обикновено с контроли като **`TextBox`** и **`CheckBox`**, които показват единична стойност.
- Сложно свързване (complex binding) – свързване на списъчна контрола със списък. Такова свързване използваме с контроли като **`ListBox`**, **`ComboBox`** и **`DataGrid`**. При него се поддържа текущо избран (активен) елемент от списъка.

Просто свързване

Чрез следващите фрагменти код ще илюстрираме как се осъществява просто свързване на данни в зависимост от източника на данни.

Свързване на контрола към обект

Нека имаме клас **Customer**, който има свойство **Name** и **TextBox** контрола с име **TextBoxName**. Свързването на свойството **Text** на **TextBox** контролата към свойството **Name** на обект от тип **Customer** се извършва по следния начин:

```
class Customer
{
   private string mName;
   public string Name
   {
      get { return mName; }
      set { mName = value; }
   }
}

Customer cust = new Customer();
cust.Name = "Бай Иван";

TextBoxName.DataBindings.Add(new Binding("Text", cust, "Name"));
```

Използвахме колекцията **DataBindings** на класа **Control**. В нея можем да добавяме **Binding** обекти, които указват кое свойство на текущата контрола с кое свойство на дадена друга контрола да бъде свързано. В нашия случай при промяна на **TextBoxName.Text** ще се променя и свойството **Name** на свързания обект **cust**.

Свързване на контрола към списък

Нека имаме масив **towns**, съдържащ имена на градове, и **TextBox** контрола с име **TextBoxTowns**. Свързването на свойството **Text** на **TextBox** контролата към масива с имена на градове се извършва по следния начин:

```
string[] towns = {"София", "Пловдив", "Варна"};
TextBoxTowns.DataBindings.Add(new Binding("Text", towns, ""));
```

Оставили сме пътя до свойството на източника, към което ще се свържем, да е празен низ, защото в случая искаме да свържем свойството **Text** директно с елементите на масива, който използваме като източник на данни, а не с тяхно свойство. При това свързване текстовата контрола ще се свърже първоначално с първия елемент от масива (символния низ "София"), но след това програмно може да се укаже промяна на текущия елемент и свързването да се промени към някой друг от елементите на

масива. На начините за промяна на текущия елемент на свързването ще се спрем след малко.

Свързване на контрола към таблица

Нека имаме **DataSet** обект **ds** с таблица **Towns** с колони **id** и **name** и **TextBox** контрола с име **TextBoxTowns**. Свързването на свойството **Text** на **TextBox** контролата към колоната **name** от таблицата може да се извърши по следния начин:

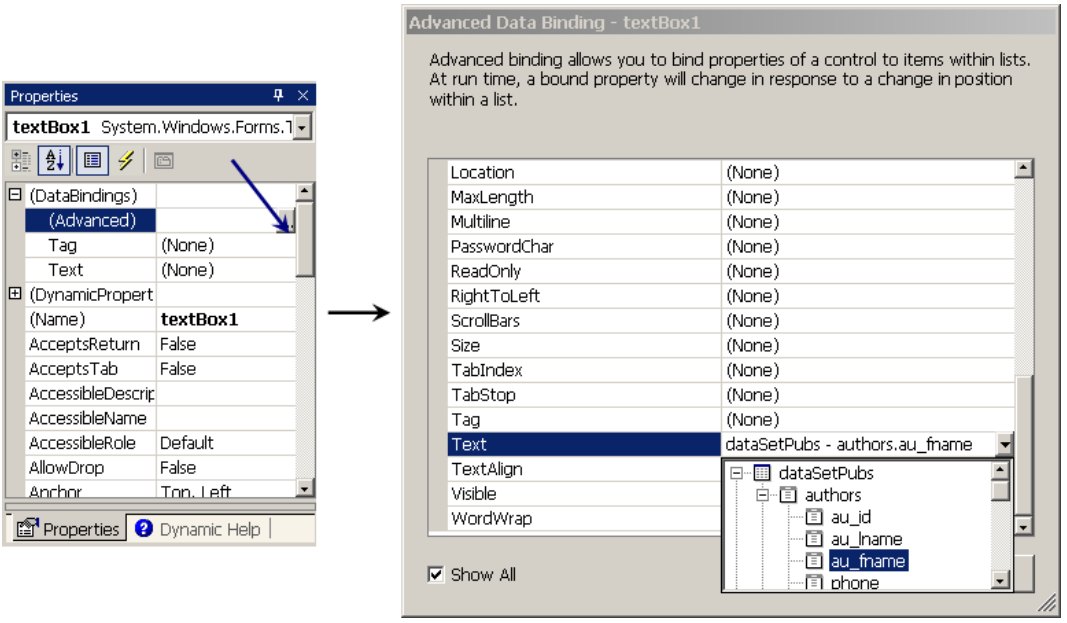
```
// Create a DataTable with columns id and name
DataTable towns = new DataTable("Towns");
towns.Columns.Add(new DataColumn("id", typeof(int)));
towns.Columns.Add(new DataColumn("name", typeof(string)));

// Add three rows to the table
DataRow row;

row = towns.NewRow();
row["id"] = 1;
row["name"] = "София";
towns.Rows.Add(row);

row = towns.NewRow();
row["id"] = 2;
row["name"] = "Пловдив";
towns.Rows.Add(row);

row = towns.NewRow();
row["id"] = 3;
row["name"] = "Варна";
towns.Rows.Add(row);

// Create a DataSet and add the table to the DataSet
DataSet ds = new DataSet();
ds.Tables.Add(towns);

TextBoxTowns.DataBindings.Add(
   new Binding("Text", ds, "Towns.name"));
```

За да укажем, че искаме да свържем свойството **Text** на **TextBox** контролата с колоната **name** на таблицата **Towns** от източника на данни **ds**, задаваме **"Towns.name"** за път до свойството на източника. Текстовото поле ще бъде свързано първоначално с първия ред на таблицата, и по-точно с полето **name** от този ред, но по-късно текущият ред може да бъде променен програмно.

Просто свързване с VS.NET

Свързването може да става и по време на дизайн в редактора на VS.NET, ако за източник на данни използваме **DataSet**. За целта от прозореца

Properties на редактора избираме **Databindings | Advanced**. Появява се прозорецът **Advanced Data Binding**. В него виждаме списък от свойствата на контролата. Намираме свойството, за което искаме да добавим свързване, и от падащия списък в дясно от него избираме източника на данни.



Свързване на контрола към обект – пример

С настоящия пример ще илюстрираме простото свързване (simple binding) в Windows Forms. За целта ще създадем просто приложение, в което ще свържем свойство на контрола със свойство на даден обект.

Ето стъпките за изграждане на нашето приложение:

1. Стартираме VS.NET и създаваме нов Windows Forms проект.
2. Задаваме на главната форма име **MainForm** и заглавие "**Binding Control To Object**". Променяме и името на файла от **Form1.cs** на **MainForm.cs**.
3. Дефинираме клас **Customer**, с чийто обект ще свържем по-късно контролата. Класът има свойство **Name**, даващо достъп до името на клиента:

```
class Customer
{
   private string mName;
   public string Name
   {
      get { return mName; }
      set { mName = value; }
   }
```

```
}
```

4. Добавяме в класа **MainForm** една член-променлива **mCustomer**. С нея ще свържем текстово поле във формата.

```
private Customer mCustomer;
```

5. В главната форма поставяме една **TextBox** контрола с име **TextBoxCustomerName**, която ще свържем с **Customer** обекта и три бутона с имена **ButtonShowCustomer**, **ButtonChangeCustomer** и **ButtonRebind**. Тези бутони ще служат съответно за показване на името на клиента, за промяна на името и за извършване на свързване (data binding) на текстовото поле с **Customer** обекта.

6. В класа **MainForm** добавяме функция **RebindFormControls()**, която свързва свойството **Text** на текстовата контрола със свойството **Name** на **Customer** обекта. За целта първо свързването се изтрива (в случай, че е било вече създадено) и след това се добавя отново:

```
private void RebindFormControls()
{
   TextBoxCustomerName.DataBindings.Clear();
   TextBoxCustomerName.DataBindings.Add(
      new Binding("Text", mCustomer, "Name"));
}
```

7. Добавяме код, който при зареждане на формата (при събитие **Load** на формата) да инициализира **Customer** обекта и да го свърже с текстовата контрола:

```
private void MainForm_Load(object sender, System.EventArgs e)
{
   mCustomer = new Customer();
   mCustomer.Name = "Бай Иван";

   RebindFormControls();
}
```

8. Добавяме обработчик на събитието **Click** на **ButtonShowCustomer** бутона. В него извличаме стойността на полето **Name** на **Customer** обекта и я показваме в диалогова кутия:

```
private void ButtonShowCustomer_Click(object sender,
   System.EventArgs e)
{
   string customerName = mCustomer.Name;
   MessageBox.Show(customerName);
}
```

9. Добавяме обработчик на събитието **Click** на **ButtonChangeCustomer** бутона. В него променяме стойността на полето **Name** на **Customer** обекта:

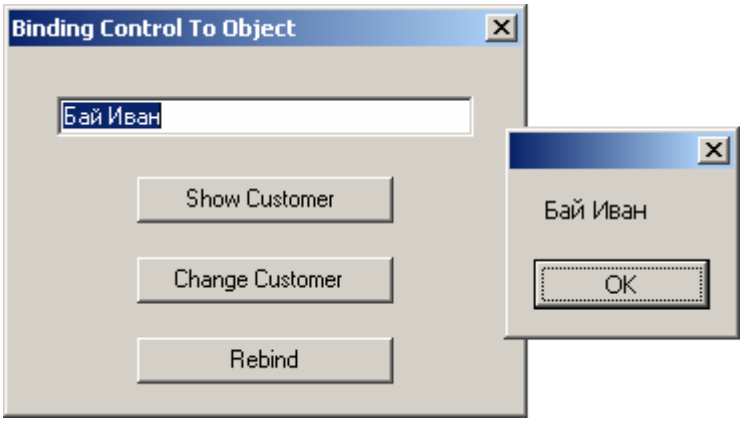
```
private void ButtonChangeCustomer_Click(object sender,
   System.EventArgs e)
{
   mCustomer.Name = "Дядо Мраз";
}
```

10. Добавяме обработчик на събитието **Click** на бутона **ButtonRebind**. В него извикваме функцията **RebindFormControls()**, която извършва повторно свързване на текстовата контрола с името на клиента от **Customer** обекта, при което това име се появява в контролата:

```
private void ButtonRebind_Click(object sender,
   System.EventArgs e)
{
   RebindFormControls();
}
```

11. Приложението вече е готово и можем да го стартираме и тестваме.

Ако въведем стойност в полето и натиснем първия бутон, в диалоговата кутия ще се покаже въведената стойност. Това показва, че стойността се е прехвърлила в **Customer** обекта:



Ако натиснем втория бутон, стойността в **Customer** обекта ще се промени. Това можем да проверим като натиснем първия бутон и изведем стойността в диалогова кутия. Въпреки че стойността в **Customer** обекта е променена, текстовото поле не се променя. Това е така, защото свързването в Windows Forms е еднопосочно – от контролата към свързвания обект, но не и обратно.

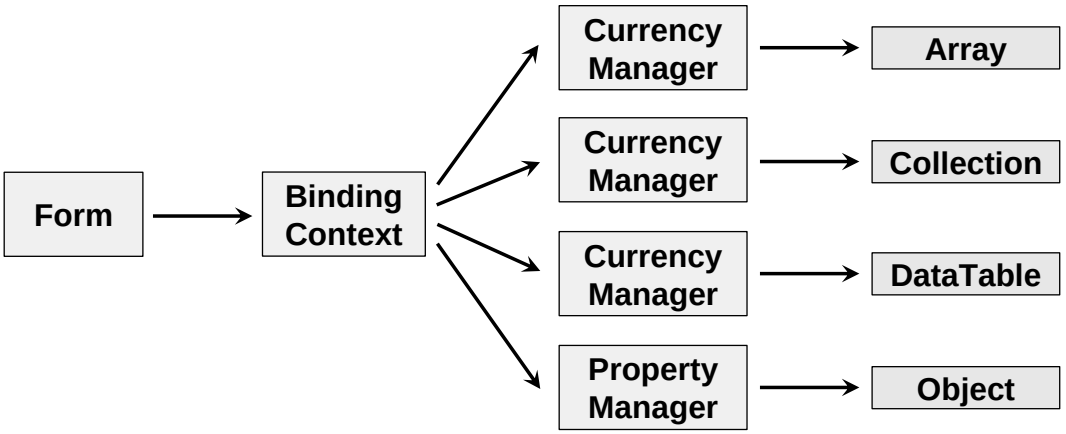
Ако сега натиснем третия бутон, текстовото поле ще се промени. Това е така, защото извършваме повторно свързване и името на клиента от **Customer** обекта се прехвърля в текстовото поле.

Binding Context

Формата пази информация за свързаните контроли в своя **BindingContext** обект. Всеки обект, наследен от класа **Control**, има един **BindingContext**, който управлява **BindingContextBase** обектите за контролите, които се съдържат в него и за самия обект. Чрез него можем да извлечем **BindingContextBase** обект за източник на данни, свързан с някоя контрола.

Понеже **BindingContextBase** е абстрактен клас, типът на върнатата стойност, в зависимост от източника на данни, е или **CurrencyManager** или **PropertyManager**, които са наследници на класа **BindingContextBase**. Ако източникът на данни е обект, който връща само една стойност (не е списък от обекти), тогава типът ще бъде **PropertyManager**. Ако източникът на данни имплементира някой от интерфейсите **IList**, **IListSource** или **IBindingList**, ще бъде върнат **CurrencyManager**.

На следващата фигура са показани схематично отношенията между Binding Context, Currency Manager и Property Manager:



Навигация с CurrencyManager

Класът **CurrencyManager** пази текущата позиция в списъка-източник на данни. Тя се съдържа в свойството **Position**. Свойството **Count** съдържа размера на списъка. Използвайки тези свойства, можем да извършваме навигация по източника на данни. За целта извличаме **CurrencyManager** обекта, свързан с източника на данни и променяме стойността на свойството **Position**.

Извличането на **CurrencyManager** обекта се извършва или чрез свойството **DataBindings** на свързаната контрола, или чрез **BindingContext** свойството на формата:

```
CurrencyManager cm = (CurrencyManager)
   textBox1.DataBindings["Text"].BindingManagerBase;

// Може и така:
CurrencyManager cm = (CurrencyManager)
   form1.BindingContext[dataTableCustomers];
```

Навигацията по списъка се извършва чрез промяна на **Position**:

```
cm.Position++;
```

Свързване на контрола към списък и навигация – пример

С настоящия пример ще илюстрираме просто свързване (simple binding) на контрола към списък и навигация по списъка чрез **CurrencyManager**.

Ето стъпките за изграждане на приложението:

1. Стартираме VS.NET и създаваме нов Windows Forms проект.
2. Задаваме на главната форма име **MainForm** и заглавие "**Binding Control To List**". Променяме и името на файла от **Form1.cs** на **MainForm.cs**.
3. Поставяме върху главната форма една **TextBox** контрола с име **TextBoxTowns**, която ще свържем с масив от символни низове - имена на градове, и два бутона с имена **ButtonPrev** и **ButtonNext**. Тези бутони ще служат съответно за навигация напред и назад по списъка с градовете. На свойството **Text** на двата бутона задаваме съответно стойности "**<< Prev**" и "**Next >>**".
4. Добавяме код, който при зареждане на формата (при събитие **Load** на формата) свързва текстовото поле с масив, съдържащ имена на градове:

```
private void MainForm_Load(object sender, System.EventArgs e)
{
   string[] towns = {"София", "Пловдив", "Варна",
      "Русе", "Бургас"};
   TextBoxTowns.DataBindings.Add(
      new Binding("Text", towns, ""));
}
```

5. Добавяме обработчик на събитието **Click** на бутона **ButtonPrev**. В него извличаме от **CurrencyManager** обекта на текстовата контрола

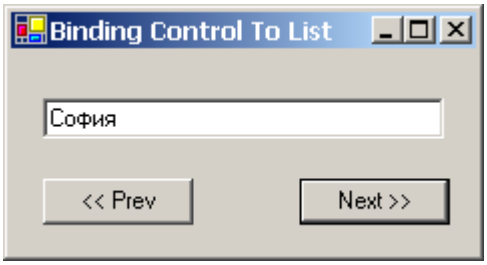
текущата позиция в списъка с градовете и я намаляваме, като, ако сме достигнали началото, позиционираме в края:

```
private void ButtonPrev_Click(object sender, System.EventArgs e)
{
   CurrencyManager cm = (CurrencyManager)
      TextBoxTowns.DataBindings["Text"].BindingManagerBase;
   if (cm.Position > 0)
   {
      cm.Position--;
   }
   else
   {
      cm.Position = cm.Count-1;
   }
}
```

6. Добавяме обработчик на събитието **Click** на бутона **ButtonNext**. В него извличаме от **CurrencyManager** на текстовата контрола текущата позиция в списъка с градовете и я увеличаваме, като, ако сме достигнали края, позиционираме в началото:

```
private void ButtonNext_Click(object sender, System.EventArgs e)
{
   CurrencyManager cm = (CurrencyManager)
      TextBoxTowns.DataBindings["Text"].BindingManagerBase;
   if (cm.Position < cm.Count-1)
   {
      cm.Position++;
   }
   else
   {
      cm.Position = 0;
   }
}
```

7. Приложението е готово и можем да го стартираме и тестваме.



При натискане на бутоните в текстовото поле ще се сменят имената на градовете. Ако променим името на някой град, промяната се отразява в масива с имената.

Сложно свързване

При сложното свързване имаме свързване на контрола към списък, като контролата се свързва с повече от един елемент от списъка. Сложното свързване се използва при списъчни контроли – **ListBox**, **ComboBox** и др.

За да свържем списъчна контрола със списък, трябва да зададем стойности на следните свойства:

- **DataSource** – източника на данни, с който ще свържем контролата.
- **DisplayMember** – път до полето, което да се визуализира.
- **ValueMember** – път до полето, от което се получава резултатът.

Стойността по подразбиране в **DisplayMember** и **ValueMember** е празен низ.

Ето как задаваме стойност на тези свойства:

```
DataSet dataSetCountries = ...;
comboBox1.DataSource = dataSetCountries;
comboBox1.DisplayMember = "Countries.CountryCode";
comboBox1.ValueMember = "Countries.Name";
```

Сложно свързване на контрола към списък – пример

С настоящия пример ще илюстрираме сложното свързване (complex data binding) в Windows Forms. За целта ще създадем просто приложение, в което ще свържем списъчна контрола със списък.

Ето стъпките за изграждане на нашето приложение:

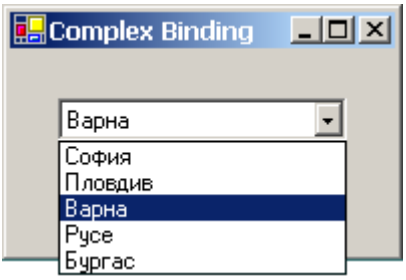
1. Стартираме VS.NET и създаваме нов Windows Forms проект.
2. Задаваме на главната форма име **MainForm** и заглавие "**Complex Binding**". Променяме името на файла от **Form1.cs** на **MainForm.cs**.
3. Поставяме във формата един бутон с име **ButtonShow** и една **ComboBox** контрола с име **ComboBoxTowns**. На свойството **Text** на бутона задаваме стойност **Show**. **ComboBox** контролата ще свържем с масив от символни низове - имена на градове, а чрез бутона ще показваме стойността, избрана в нея.
4. Добавяме код, който при зареждане на формата (при събитие **Load**) свързва **ComboBox** контролата с масив, съдържащ имена на градове:

```
private void MainForm_Load(object sender, System.EventArgs e)
{
  string[] towns = {"София", "Пловдив", "Варна",
    "Русе", "Бургас"};
  ComboBoxTowns.DataSource = towns;
  ComboBoxTowns.DisplayMember = "";
}
```

5. Добавяме обработчик на събитието **Click** на бутона **ButtonShow**. В него показваме в диалогова кутия стойността, избрана в **ComboBox** контролата:

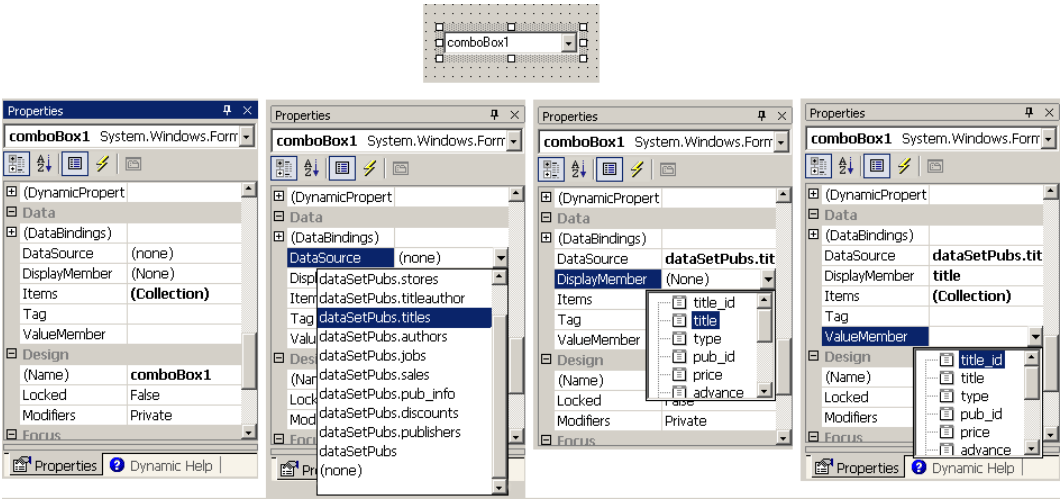
```
private void ButtonShow_Click(object sender, System.EventArgs e)
{
  MessageBox.Show(ComboBoxTowns.SelectedValue.ToString());
}
```

6. Приложението ни е готово и можем да го стартираме и тестваме.



Сложно свързване с VS.NET

Сложното свързване може да става и по време на дизайн в редактора на VS.NET, ако за източник на данни използваме **DataSet**. За целта в прозореца **Properties** на редактора щракваме върху падащия списък от дясно на свойството **DataSource** и избираме от него източник на данни. След това избираме от падащите списъци в дясно от свойствата **DisplayMember** и **ValueMember** полето, което ще се визуализира, и полето, от което ще се получава резултатът:



Контролата DataGrid

DataGrid контролата визуализира таблични данни. Тя осигурява навигация по редове и колони и позволява редактиране на данните. Като източник на данни най-често се използват ADO.NET **DataSet** и **DataTable**. Чрез свойството **DataSource** се задава източникът на данни, а чрез свойството **DataMember** – пътят до данните в рамките на източника. По-важни свойства на контролата са:

- **ReadOnly** – разрешава / забранява редакцията на данни.
- **CaptionVisible** – показва / скрива заглавието.
- **ColumnHeadersVisible** – показва / скрива заглавията на колоните.
- **RowHeadersVisible** – показва / скрива колоната в ляво от редовете.
- **TableStyles** – задава стилове за таблицата.
  - **MappingName** – задава таблицата, за която се отнася дефинираният стил.
  - **GridColumnStyles** – задава форматиране на отделните колони – заглавие, ширина и др.

Противно на очакванията контролата **DataGrid** няма събитие "смяна на текущия избран ред". Ако ви се налага да извършвате някакво действие при смяна на текущия избран ред (например да запишете промените по текущия ред в базата данни), можете да прихванете събитието **CurrentCellChanged**, което се активира при промяна на текущата клетка. Ако запомните в член-променлива в класа на формата коя е била предишната текуща клетка, ще можете да проверите дали текущият ред е бил променен. Текущата клетка е достъпна от свойството **CurrentCell**.

Препоръчителен начин за използване на **DataGrid** контролата е в режим **ReadOnly=true**. В този случай не се разрешават директни промени, а това спестява много проблеми. Ако е необходимо редактиране на редове или добавяне на нови, това може да се направи с отделен диалогов прозорец, който излиза при натискане на бутон "**Edit**" или "**Add**" при избран ред от таблицата.

Работа с DataGrid контролата – пример

С настоящия пример ще илюстрираме работата с **DataGrid** контролата в Windows Forms и сложното свързване (complex data binding) на таблица от **DataSet** с **DataGrid**.

Ето стъпките за изграждане на нашето приложение:

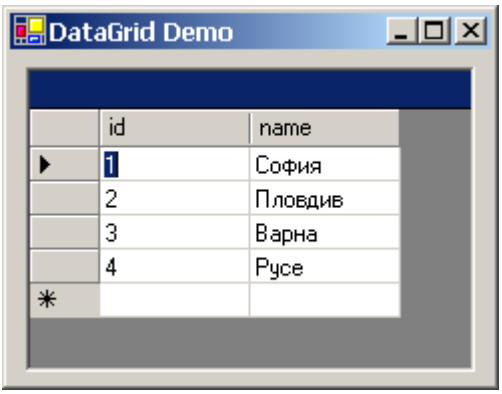
1. Стартираме VS.NET и създаваме нов Windows Forms проект.
2. Задаваме на главната форма име **MainForm** и заглавие "**DataGrid Demo**". Променяме и името на файла от **Form1.cs** на **MainForm.cs**.

3. Поставяме във формата една **DataGrid** контрола. За име на контролата задаваме **DataGridTowns**.

4. Добавяме код, който при зареждане на формата (при събитие **Load**) създава **DataSet**, съдържащ таблица **Towns** с две колони – **id** и име на град. След като той е създаден, свързваме **DataGrid** контролата с таблицата **Towns** от този **DataSet**:

```
private void MainForm_Load(object sender, System.EventArgs e)
{
  // Create table "Towns"
  DataTable towns = new DataTable("Towns");
  towns.Columns.Add(new DataColumn("id", typeof(int)));
  towns.Columns.Add(new DataColumn("name", typeof(string)));

  // Add some rows in the table
  DataRow row = towns.NewRow();
  row["id"] = 1;
  row["name"] = "София";
  towns.Rows.Add(row);

  row = towns.NewRow();
  row["id"] = 2;
  row["name"] = "Пловдив";
  towns.Rows.Add(row);

  row = towns.NewRow();
  row["id"] = 3;
  row["name"] = "Варна";
  towns.Rows.Add(row);

  row = towns.NewRow();
  row["id"] = 4;
  row["name"] = "Русе";
  towns.Rows.Add(row);

  // Add table "Towns" to the DataSet
  DataSet ds = new DataSet();
  ds.Tables.Add(towns);

  // Bind the DataGrid to the DataSet
  DataGridTowns.DataSource = ds;
  DataGridTowns.DataMember = "Towns";
}
```

5. Приложението е готово и можем да го стартираме и тестваме.



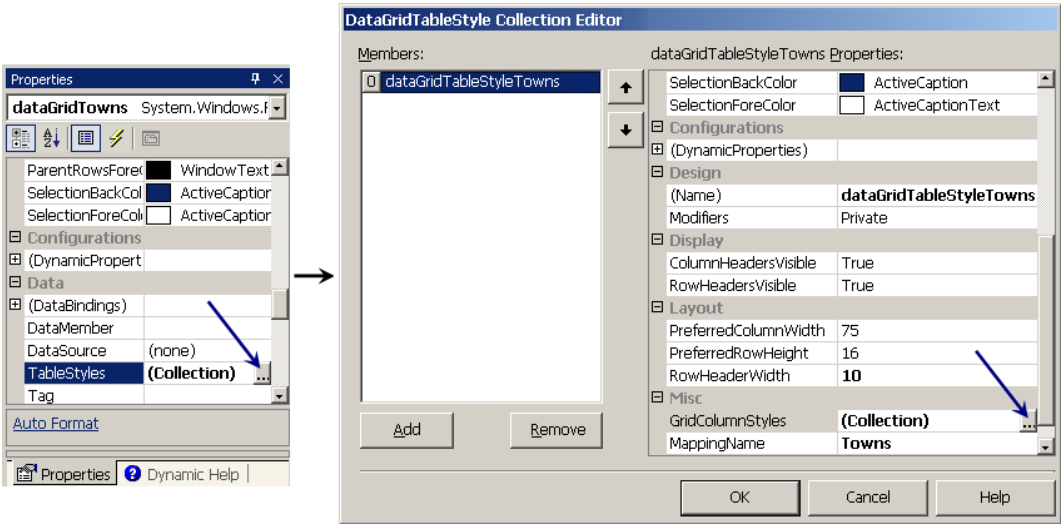
Ако променим данните, визуализирани в **`DataGrid`** контролата, те ще се променят и в таблицата **`Towns`** от **`DataSet`** обекта.

TableStyles и дефиниране на стилове – пример

Настоящият пример илюстрира работата с **`DataGrid`** контролата в Windows Forms и възможностите за дефиниране на стилове за визуализацията на данните чрез колекцията **`TableStyles`**. Ще създадем просто приложение, подобно на това от предходния пример, но чрез колекцията **`TableStyles`** ще определим как да бъдат визуализирани колоните на таблицата.

Ето и стъпките за изграждане на нашето приложение:

1. Началните стъпки за изграждане на приложението са същите като стъпки от 1 до 4 в предишния пример. Изпълняваме ги и преминаваме към дефинирането на стиловете за визуализация на данните.

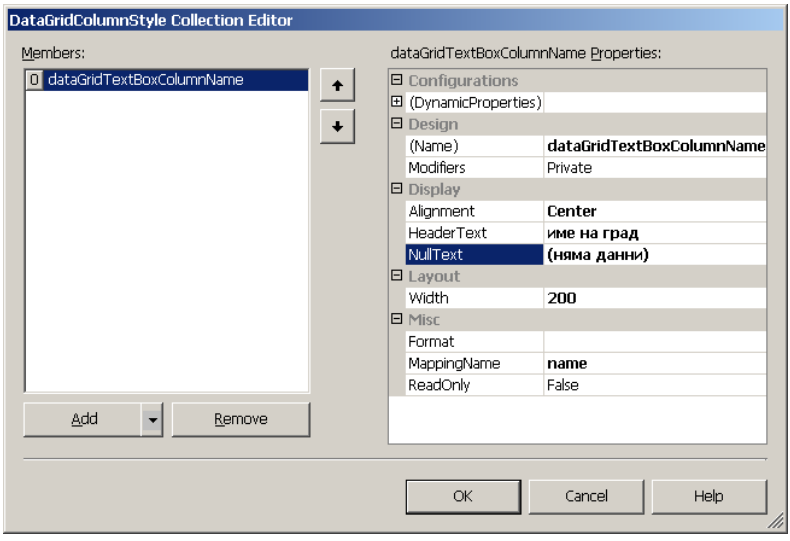


2. Щракваме с десния бутон на мишката върху поставения в главната форма **`DataGrid`** и избираме **`Properties`**. В прозореца **`Properties`** на редактора избираме свойството **`TableStyles`** и щракваме върху

бутона с многоточието, намиращ се в полето в дясно от него. Отваря се прозорец, който ни позволява да добавяме стилове за таблицата. Щракваме върху бутона **Add**, за да добавим нов стил. В дясната половина на прозореца можем да променяме свойствата на добавения стил. Намираме свойството **Name** и му задаваме стойност **DataGridTableStyleTowns**.

3. На свойството **MappingName** задаваме стойност **Towns**. С това указваме, че този стил се отнася за таблицата **Towns**. Задаваме на свойството **AlternatingBackColor** (указващо цвят, в който ще се оцветяват четните редове) стойност **Info**. Остана да зададем стилове за отделните колони.

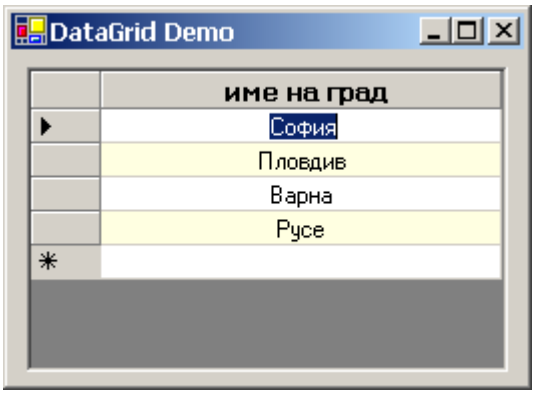
4. Щракваме върху бутона с многоточието, намиращ се в полето в дясно от свойството **GridColumnStyles**. Отваря се прозорец, който ни позволява да добавяме стилове за отделните колони. Щракваме върху бутона **Add**, за да добавим нов **DataGridTextBoxColumn** в колекцията. Задаваме стойност **DataGridTextBoxColumnName** на свойството **Name**.



5. Задаваме на свойството **MappingName** стойност **name**. Така указваме, че този стил се отнася за полето **name**. Задаваме на свойствата **Alignment**, **HeaderText** и **NullText** съответно стойности **Center**, "**име на град**" и "**(няма данни)**". Така заглавието на колоната ще е "**име на град**", текстът ще е центриран, а когато няма стойност в полето, в таблицата ще се визуализира "**(няма данни)**". Накрая указваме ширина на колоната, като на свойството **width** зададем стойност **200**.

6. Натискаме бутона **[OK]**, за да запазим промените в колекцията със стиловете за колоните. След това натискаме бутона **[OK]** и в другия прозорец, за да запазим промените в стиловете за таблиците.

7. Приложението е готово и можем да го стартираме и тестваме.



Забелязваме, че макар в таблицата **`Towns`** да има две колони, в нашия **`DataGrid`** се визуализира само едната. Това е така, защото се визуализират само полетата, за които са добавени стилове в колекцията **`GridColumnStyles`**. Това означава, че ако не искаме дадено поле да бъде визуализирано, просто не указваме стил за него.

Ще отбележим, че когато добавяме стил в колекцията **`GridColumnStyles`**, освен **`DataGridTextBoxColumn`**, можем да добавяме и **`DataGridBoolColumn`**. Това става, като щракнем върху стрелката, намираща се в дясната част на бутона **`Add`**, и от падащия списък изберем **`DataGridBoolColumn`**. Чрез **`DataGridBoolColumn`** определяме колона, която във всяка клетка съдържа поле с отметка, представящо булева стойност.

Master-Details навигация

Навигацията "главен/подчинен" (master-details) отразява взаимоотношения от тип едно към много (например един регион има много области). В ADO.NET **`DataSet`** обектите поддържат релации от тип "главен/подчинен". За целта се използват **`DataRelation`** обектите в **`DataSet`**.

В Windows Forms се поддържа навигация "главен/подчинен". За да илюстрираме работата с нея, нека разгледаме един пример: Имаме **`DataSet`**, съдържащ две таблици – едната съдържа имена на държави, а другата – имена на градове. Те са свързани помежду си така, че на всяка държава от първата таблица съответстват определени градове от втората таблица:

Тогава можем да използваме две **`DataGrid`** контроли – първата, визуализираща държавите, а втората, визуализираща градовете, съответстващи

на текущо избраната държава от първата контрола. За целта контролите се свързват с един и същ **DataSet**. На главната контрола се задава за източник на данни главната таблица. На подчинената контрола се задава за източник на данни релацията на таблицата:

```
// Bind the master grid to the master table
DataGridCountries.DataSource = datasetCountriesAndTowns;
DataGridCountries.DataMember = "Countries";

// Bind the detail grid to the relationship
DataGridTowns.DataSource = datasetCountriesAndTowns;
DataGridTowns.DataMember = "Countries.CountriesTowns";
```

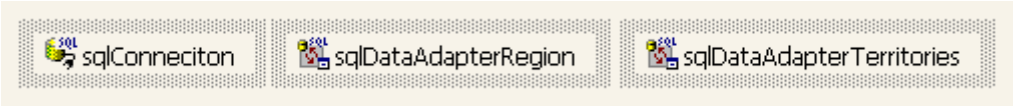
Master-Details навигация – пример

Настоящият пример илюстрира възможностите за реализация на Master-Details навигация, базирана на **DataSet** компонентата от ADO.NET и сложното свързване на списъчни контроли в Windows Forms. В примера ще използваме базата данни **Northwind** – една от стандартните демонстрационни бази в MS SQL Server.

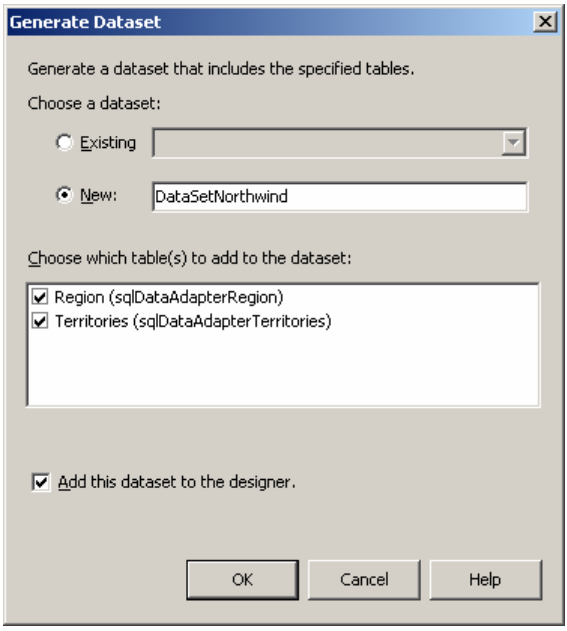
Ще създадем приложение, което има в главната си форма две контроли – **ListBox**, който показва региони (от таблицата **Region** от базата данни), и **DataGrid**, който показва областите за всеки регион (от таблицата **Territories** от базата данни).

Ето и стъпките за изграждане на нашето приложение:

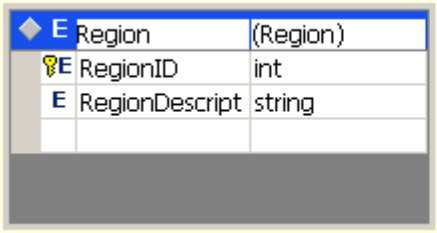
1. Стартираме VS.NET и създаваме нов Windows Forms проект.
2. Задаваме на главната форма име **MainForm** и заглавие "**Master-Detail Demo**". Променяме и името на файла от **Form1.cs** на **MainForm.cs**.
3. В прозореца Server Explorer от VS.NET намираме демонстрационната база данни **Northwind** на MS SQL Server. Щракваме върху таблицата **Region** и след това, натискайки клавиш **Ctrl**, щракваме върху таблицата **Territories**. След като сме маркирали едновременно и двете таблици, ги извличаме върху формата. Ако прозорецът Server Explorer не е отворен, можем да го отворим, като изберем **View | Server Explorer**.
4. Windows Forms редакторът автоматично създава за нас един **SqlConnection** и два **SqlDataAdapter** компонента. Променяме техните имена съответно на **sqlConneciton**, **sqlDataAdapterRegion** и **sqlDataAdapterTerritories**:

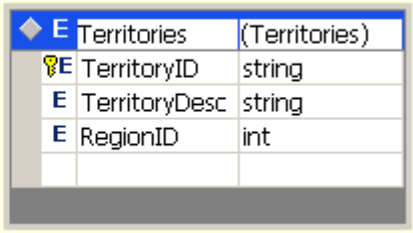


5. От менюто **Data** избираме **Generate Dataset**... В появилия се прозорец указваме, че искаме да създадем нов **DataSet** и задаваме за име **DataSetNorthwind**. Поставяме отметки и пред двете таблици и натискаме бутона **[OK]**, за да създадем новия **DataSet**. Променяме името на появилия се в редактора **DataSet** на **dataSetNorthwind**.

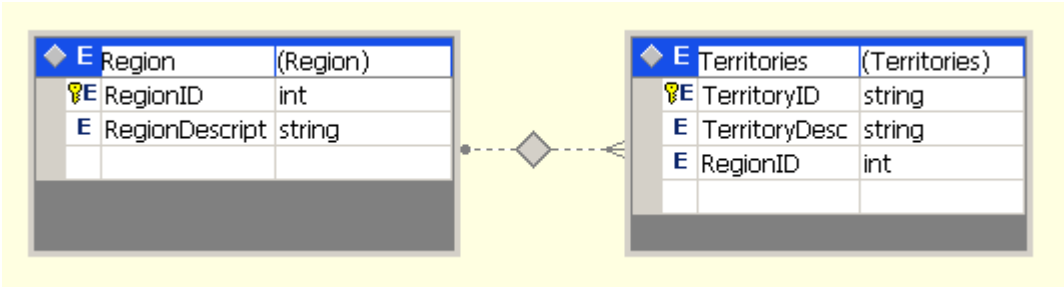


6. Щракваме с десния бутон върху **dataSetNorthwind** в редактора и от появилото се контекстно меню избираме **View Schema**... Отваря се файлът **DataSetNorthwind.xsd** - виждаме XSD схемата на **DataSet**-а, генериран на базата на таблиците **Region** и **Territories**.





7. От **Toolbox** извличаме един **Relation** обект и го пускаме върху таблицата **Territories**. В появилия се прозорец се уверяваме, че за **Parent element** е избрана таблицата **Region**, а за **Child element** - таблицата **Territories**, и натискаме бутона **OK**. Така дефинирахме релация тип Master-Details между таблиците **Region** и **Territories**.



8. Добавяме във формата един **ListBox** с име **ListBoxRegions**. На свойството **DataSource** задаваме стойност **dataSetNorthwind**, а на свойствата **DisplayMember** и **ValueMember** – съответно стойности **Region.RegionDescription** и **Region.RegionID**.

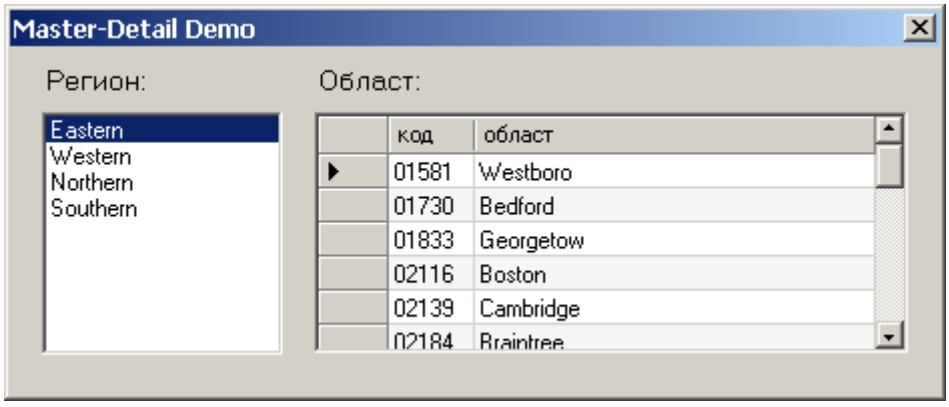
9. Добавяме във формата един **DataGrid** с име **DataGridTerritories**. Задаваме на свойствата **DataSource** и **DataMember** съответно стойности **dataSetNorthwind** и **Region.RegionTerritories**.

10. Дефинираме стил с име **dataGridTableStyleTerritories** за таблицата **Territories**. В колекцията му **GridColumnStyles** добавяме стилове за полетата **TerritoryID** и **TerritoryDescription**, като указваме, че тези колони трябва да са със заглавия - съответно **код** и **област**.

11. Добавяме код, който при зареждане на формата (при събитие **Load**) зарежда **DataSet** обекта от базата данни чрез **DataAdapter** компонентите за двете таблици (**Region** и **Territories**):

```
private void MainForm_Load(object sender, System.EventArgs e)
{
   sqlDataAdapterRegion.Fill(dataSetNorthwind);
   sqlDataAdapterTerritories.Fill(dataSetNorthwind);
}
```

12. Приложението е готово и можем да го стартираме и тестваме:



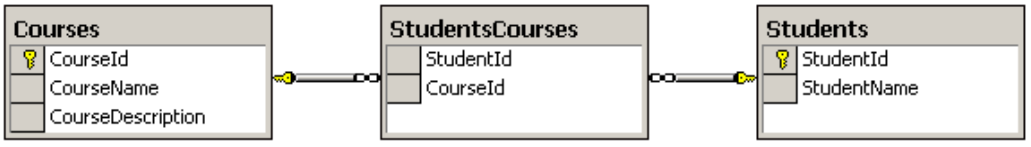
Проблеми при Master-Details навигацията

Показаният начин за реализация на master-details навигация е лесен за използване, но има един сериозен проблем: винаги зарежда в паметта всички записи от двете таблици. Ако таблиците са обемни, този подход ще работи много бавно или въобще няма да работи. Причината е, че зареждането на голям обем записи (да кажем няколко хиляди) в **DataSet** изисква много памет и става бавно.

Ако данните са много, можем да подходим по следния начин: Зареждаме всички данни от главната (master) таблица и ги визуализираме с **DataGrid** или **ListBox**. След това прихващаме събитието "смяна на текущия ред" и при неговото настъпване зареждаме в подчинената (details) таблица детайлните записи за избрания запис от главната таблица. Зареждането може да се извърши с параметрична SQL заявка, изпълнена през **SqlDataReader** или **SqlDataAdapter**.

Релации "много към много"

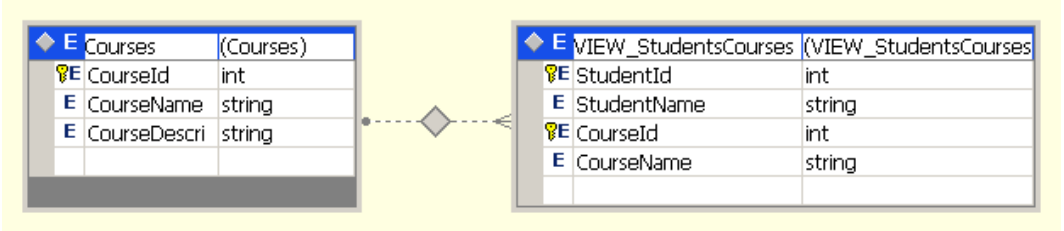
DataSet и **DataGrid** не поддържат релации тип "много към много". Такъв тип релации могат да бъдат сведени до Master-Details чрез добавяне на изглед в базата данни. Нека примерно имаме база данни, съдържаща таблици **Courses** и **Students** и таблица **StudentsCourses**, осъществяваща връзка между тях.



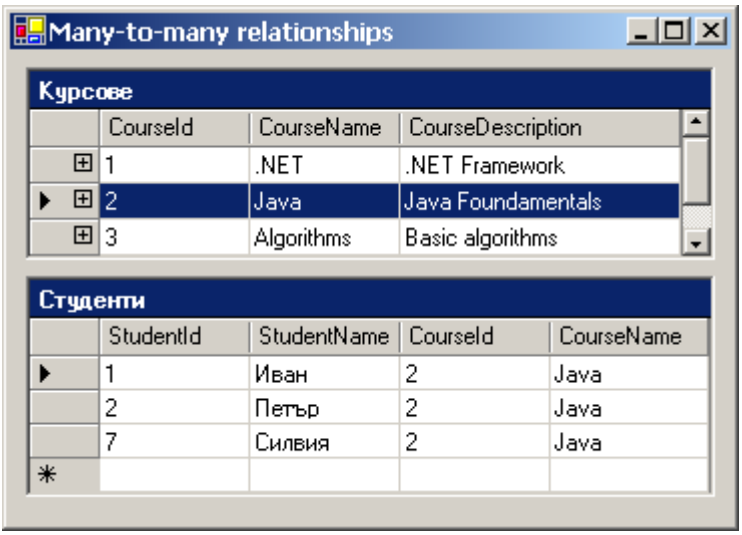
За да сведем тази релация към Master-Details, можем да създадем изглед в базата данни:

```
CREATE VIEW View_StudentsCourses AS
   SELECT StudentId, StudentName, CourseId, CourseName
   FROM Students, Courses, StudentsCourses
   WHERE Students.StudentsId = StudentsCourses.StudentId
   AND Courses.CourseId = StudentsCourses.CourseId
```

След като сме създали изгледа, можем да сведем релацията до релация Master-Details между таблицата **Courses** и новосъздадения изглед:



Аналогично на предходния пример можем да работим с таблиците, които са вече във взаимоотношение "главен/подчинен":



Наследяване на форми

Наследяването на форми позволява повторно използване на части от потребителския интерфейс. Чрез него е възможно да променим наведнъж общите части на много форми. За целта дефинираме една базова форма, която съдържа общата за всички наследници функционалност.

Базовата форма е най-обикновена форма. Единствената особеност е, че контролите, които могат да се променят от наследниците, се обявяват като **`protected`**. Виртуални методи могат да реализират специфичната за наследниците функционалност, достъпна от базовата форма.

При наследяване на форма се наследява класът на базовата форма. При това се указва името на пространството, в което е дефинирана базовата форма, следвано от точка, и името на базовата форма. Във Visual Studio .NET формите наследници могат да се създават, като от менюто се избере **`File | Add New Item… | Inherited Form`**.

При наследяването на форми можем да поставим базовата форма и формите-наследници в различни асемблита и след това да променяме всички форми-наследници чрез промяната на единичен DLL файл.

Една особеност на VS.NET е, че по време на дизайн промените, направени върху базовата форма, не се отразяват върху формите наследници, преди да бъде прекомпилирано приложението.

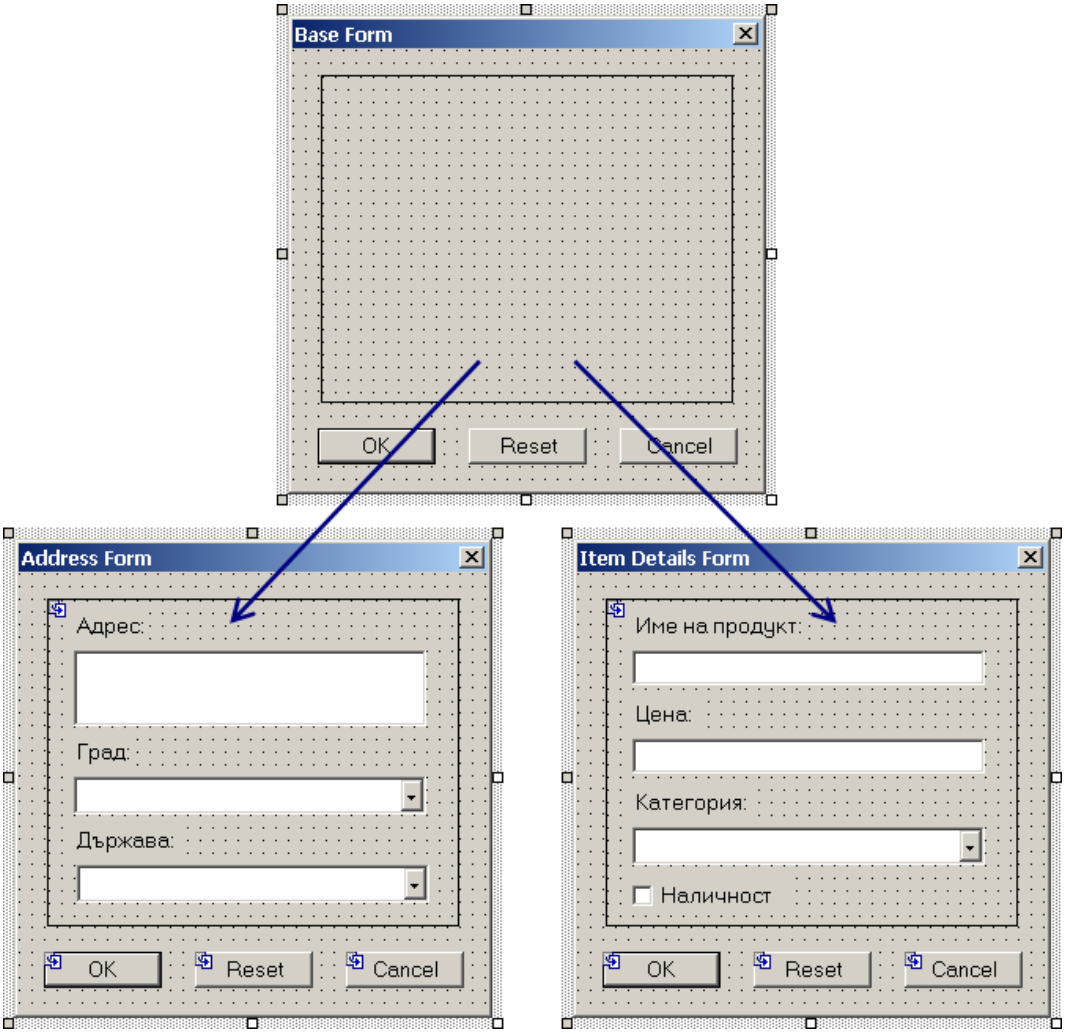
Наследяване на форми – пример

Настоящият пример илюстрира възможностите за наследяване на форми в Windows Forms, при което се наследяват всички контроли в тях, както и

методите и свойствата на класа, в който са дефинирани. В примера ще създадем четири форми:

- **`MainForm`** – главната форма на приложението, която ще служи за показване на другите форми при натискане на съответния бутон.
- **`BaseForm`** – базова форма, от която други форми наследяват потребителски интерфейс и базова функционалност.
- **`AddressForm`** – форма за попълване на адрес, наследник на **`BaseForm`**.
- **`ItemsDetailsForm`** – форма за попълване на описание на продукт, наследник на **`BaseForm`**.

Схематично наследяването между формите е показано на фигурата по-долу:



Ето и стъпките за изграждане на нашето приложение:

1. Стартираме VS.NET и създаваме нов Windows Forms проект.
2. Задаваме на главната форма име **MainForm** и заглавие "**Main Form**". Променяме и името на файла от **Form1.cs** на **MainForm.cs**.
3. Добавяме нова форма с име **BaseForm**. Това ще бъде нашата базова форма. От нея ще наследим останалите форми. В нея поставяме един **Panel** с име **PanelMain** и три бутона **ButtonOK**, **ButtonCancel** и **ButtonReset**. Дефинираме панела като **protected**, за да може да се променя от наследниците. Бутоните **ButtonOK** и **ButtonCancel** имат обичайното предназначение, което е зададено със свойствата **AcceptButton** и **CancelButton** на формата.
4. Добавяме обработчик на събитието **Click** на бутона **ButtonReset**. В него ще извикваме виртуалния метод **ResetFormFields()**, който трябва да се имплементира в наследниците и трябва да изтрива всички полета:

```
private void ButtonReset_Click(object sender,System.EventArgs e)
{
  ResetFormFields();
}
```

5. Добавяме и виртуалния метод **ResetFormFields()**:

```
protected virtual void ResetFormFields()
{
  // Descendand form should implement reset functionality here
}
```

6. Компилираме приложението, за да се създаде асемблито, в което ще се съдържа формата, от която ще наследяваме. За целта избираме **Build | Build Solution**.
7. Добавяме първата форма-наследник. За целта избираме **File | Add New Item… | Inherited Form**. Въвеждаме за име на формата **AddressForm** и натискаме бутона **open**. От появилия се списък избираме **BaseForm** за компонент, от който ще наследяваме. Отваря се формата-наследник, която изглежда точно като базовата форма.
8. Променяме заглавието ѝ на **Address Form**. Добавяме във формата един **TextBox** с име **TextBoxAddress** и две **ComboBox** контроли с имена **ComboBoxTown** и **ComboBoxCountry**. Задаваме на свойството **Multiline** на **TextBoxAddress** стойност **true**. За **DropDownStyle** на **ComboBox** контролите задаваме **DropDownList**. В колекцията **Items** на **ComboBoxTowns** въвеждаме няколко имена на български градове, а в тази на **ComboBoxTowns** въвеждаме "**България**".

9. Предефинираме метода **ResetFormFields()** така, че да изчиства полетата на формата:

```
protected override void ResetFormFields()
{
   this.TextBoxAddress.Clear();
   this.ComboBoxTown.SelectedIndex = -1;
   this.ComboBoxCountry.SelectedIndex = -1;
}
```

10. Добавяме втората форма-наследник. Задаваме **ItemDetailsForm** за име на формата. Променяме заглавието ѝ на **Item Details Form**. Добавяме във формата две **TextBox** контроли с имена **TextBoxName** и **TextBoxPrice**, един **ComboBox** с име **ComboBoxCategory** и един **CheckBox** с име **ChackBoxAvailability**. За **DropDownStyle** на **ComboBoxCategory** задаваме **DropDownList**, а в колекцията **Items** въвеждаме няколко категории – "**Алкохол**", "**Безалкохолни напитки**", "**Колбаси**", "**Стоки за бита**". Задаваме на свойството **Text** на **ChackBoxAvailability** стойност "**Наличност**".

11. Предефинираме и в тази форма метода **ResetFormFields()** така, че да изчиства полетата:

```
protected override void ResetFormFields()
{
   this.TextBoxName.Clear();
   this.TextBoxPrice.Clear();
   this.ComboBoxCategory.SelectedIndex = -1;
   this.CheckBoxAvailability.Checked = false;
}
```

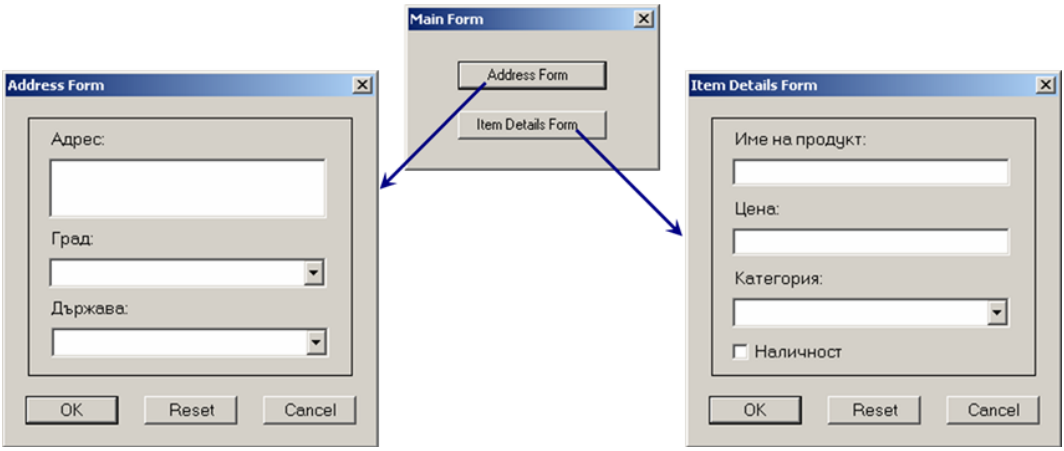
12. Поставяме подходящи етикети на контролите във формите – например при **TextBoxName** поставяме етикет, чието свойство **Text** има стойност "**Име на продукт:**".

13. В главната форма добавяме два бутона с имена **ButtonAddressForm** и **ButtonItemDetailsForm**. В обработчиците на събитията **Click** на тези бутони ще показваме формите наследници:

```
private void ButtonAddressForm_Click(object sender,
   System.EventArgs e)
{
   AddressForm addressForm = new AddressForm();
   addressForm.ShowDialog();
   addressForm.Dispose();
}

private void ButtonItemDetailsForm_Click(object sender,
   System.EventArgs e)
```

```
{
   ItemDetailsForm itemDetailsForm = new ItemDetailsForm();
   itemDetailsForm.ShowDialog();
   itemDetailsForm.Dispose();
}
```

14. Приложението е готово и можем да го стартираме и тестваме:



Пакетът System.Drawing и GDI+

Пакетът **`System.Drawing`** осигурява достъп до GDI+ функциите на Windows:

- повърхности за чертане
- работа с графика и графични трансформации
- изчертаване на геометрични фигури
- работа с изображения
- работа с текст и шрифтове
- печатане на принтер

Той се състои от няколко пространства:

- **`System.Drawing`** – предоставя основни класове като повърхности, моливи, четки, класове за изобразяване на текст.
- **`System.Drawing.Imaging`** – предоставя класове за работа с изображения, картинки и икони, класове за записване в различни файлови формати и за преоразмеряване на изображения.
- **`System.Drawing.Drawing2D`** – предоставя класове за графични трансформации – бленди, матрици и др.
- **`System.Drawing.Text`** – предоставя класове за достъп до шрифтовете на графичната среда.

- **System.Drawing.Printing** – предоставя класове за печатане на принтер и системни диалогови кутии за печатане.

Класът Graphics

Класът **System.Drawing.Graphics** предоставя абстрактна повърхност за чертане. Такава повърхност може да бъде както част от контрола на екрана, така и част от страница на принтер или друго устройство.

Най-често чертането се извършва в обработчика на събитието **Paint**. В него при необходимост се преизчертава графичния облик на контролата. Параметърът **PaintEventArgs**, който се подава, съдържа **Graphics** обекта. **Graphics** обект може да се създава чрез **Control.CreateGraphics()**. Той задължително трябва да се освобождава чрез **finally** блок или с конструкцията **using**, защото е ценен ресурс.

Работа със System.Drawing – пример

Чрез настоящия пример ще илюстрираме работата с GDI+ чрез пакета **System.Drawing** – чертане на геометрични фигури с четки и моливи и изобразяване на текст със зададен шрифт.

Ето и стъпките за изграждане на нашето примерно приложение:

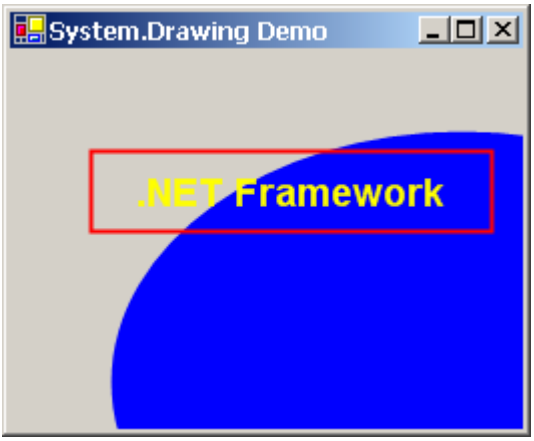
1. Стартираме VS.NET и създаваме нов Windows Forms проект.
2. Задаваме на главната форма име **MainForm** и подходящо заглавие, например "**System.Drawing Demo**". Променяме и името на файла от **Form1.cs** на **MainForm.cs**.
3. Добавяме обработчик на събитието **Paint**, където изчертаваме графично изображение:

```
private void MainForm_Paint(object sender,
   System.Windows.Forms.PaintEventArgs e)
{
   Graphics g = e.Graphics;
   g.SmoothingMode = SmoothingMode.AntiAlias;

   Brush brush = new SolidBrush(Color.Blue);
   g.FillEllipse(brush, 50, 40, 350, 250);
   brush.Dispose();

   Pen pen = new Pen(Color.Red, 2);
   g.DrawRectangle(pen, 40, 50, 200, 40);
   pen.Dispose();

   brush = new SolidBrush(Color.Yellow);
   Font font = new Font("Arial", 14, FontStyle.Bold);
   g.DrawString(".NET Framework", font, brush, 60, 60);
   brush.Dispose();
```

```
    font.Dispose();
}
```

В метода вземаме **Graphics** обекта на формата, създаваме подходящи четки, моливи и шрифтове. С тях изчертаваме запълнена елипса и правоъгълник и в него изписваме текст. Всички GDI+ ресурси (четки, моливи и шрифтове) задължително се освобождават след използване.

4. Приложението е готово и можем да го стартираме и тестваме:



Анимация със System.Drawing – пример

Настоящият пример илюстрира как със средствата на GDI+ чрез пакета **System.Drawing** може да се реализира плавна анимация на някакъв геометричен обект.

Ето и стъпките за изграждане на нашето примерно приложение:

1. Стартираме VS.NET и създаваме нов Windows Forms проект.
2. Задаваме на главната форма име **MainForm** и заглавие "**System.Drawing Demo**". Променяме и името на файла от **Form1.cs** на **MainForm.cs**.
3. Добавяме променливи и константи за позицията на анимирания обект (елипса), стъпката на преместване и размерите на елипсата:

```
private int mPosX = 0;
private int mPosY = 0;

private int StepX = 1;
private int StepY = 1;

public const int ELLIPSE_SIZE_X = 70;
public const int ELLIPSE_SIZE_Y = 40;
```

4. Поставяме в главната форма една **Timer** компонента с име **TimerAnimaiton** и един **PictureBox** с име **PictureBoxAnimatoin**.

5. Добавяме обработчик на събитието **Paint** на **PictureBox** контролата. В него изчертаваме движещия се обект на позицията, в която се намира в момента:

```
private void PictureBoxAnimation_Paint(object sender,
   System.Windows.Forms.PaintEventArgs e)
{
   Graphics g = e.Graphics;
   g.SmoothingMode = SmoothingMode.AntiAlias;

   Brush brush = new SolidBrush(Color.Blue);
   g.FillEllipse(brush, mPosX, mPosY,
      ELLIPSE_SIZE_X, ELLIPSE_SIZE_Y);
   brush.Dispose();

   brush = new SolidBrush(Color.Yellow);
   Font font = new Font("Arial", 14, FontStyle.Bold);
   g.DrawString(".NET", font, brush, mPosX+10, mPosY+10);
   brush.Dispose();
   font.Dispose();
}
```

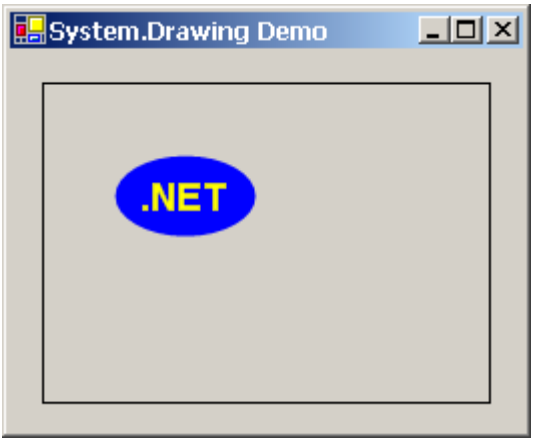
6. Задаваме на свойствата **Enabled** и **Interval** на **Timer** компонентата съответно стойности **true** и **10**. Така тя ще генерира събитие на всеки 10 милисекунди.

7. Добавяме обработчик на събитието **Elapsed** на **Timer** компонентата. В него променяме координатите на движещия се обект и пречертаваме **PictureBox** контролата:

```
private void TimerAnimation_Elapsed(object sender,
   System.Timers.ElapsedEventArgs e)
{
   mPosX += StepX;
   if ((mPosX >= PictureBoxAnimation.Width - ELLIPSE_SIZE_X - 3)
        || (mPosX <= 0))
   {
      StepX = -StepX;
   }

   mPosY += StepY;
   if ((mPosY >= PictureBoxAnimation.Height - ELLIPSE_SIZE_Y - 3)
        || (mPosY <= 0))
   {
      StepY = -StepY;
   }
```

```
   PictureBoxAnimation.Refresh();
}
```

8. Приложението е готово и можем да го стартираме и тестваме:



В примера сме използвали **PictureBox** контрола, защото тя не чертае нищо в своя **Paint** метод, който се извиква преди всяко пречертаване. Ако бяхме използвали **Panel** или друга контрола, щеше да се получи трепкане.

За професионална анимация се използва DirectX технологията, която използва ресурсите на графичната карта много по-ефективно и натоварва централния процесор много по-малко. Като цяло за по-сложни приложения (например игри) използваният в този пример подход е грешен!

Печатане на принтер

Често се налага създадените от нас приложения да отпечатват някаква информация на принтер. Пространството **System.Drawind.Printing** ни предоставя класове, чрез които можем да реализираме такава функционалност.

При печатането на принтер се използват три ключови класа:

- **PrintDialog** – стандартен диалог за печатане на принтер. Позволява на потребителя да избере принтер и да укаже кои части от документа да се отпечатат.
- **PrintController** – управлява процеса на печатане и активира събития, свързани с него. **PrintController** предоставя **Graphics** повърхността, върху която печатаме.
- **PrintDocument** – описва характеристиките на отпечатвания документ. Съдържа **PrinterSettings**, върнати от **PrintDialog**.

Обикновено, когато искаме да отпечатаме нещо на принтер, създаваме инстанция на класа **PrintDocument**, задаваме стойности на свойствата,

описващи какво ще печатаме, и извикваме метода **Print()**, за да отпечатаме документа.

Потребителски контроли

Потребителските контроли (custom controls) позволяват разширяване на стандартния набор от контроли чрез комбиниране на съществуващи контроли, разширяване на съществуващи или създаване на съвсем нови такива.

Потребителските контроли или разширяват съществуващи контроли, или класа **Control** или класа **UserControl**. Те могат да управляват поведението си по време на изпълнение, както и да взаимодействат с дизайнера на VS.NET по време на дизайн.

Създаване на нова контрола, която не наследява съществуваща

Създаването на нова контрола, която не наследява никоя съществуваща вече контрола, става по следния начин:

1. От VS.NET избираме **File | Add New Item … | UI | Custom Control**.
2. Припокриваме виртуалния метод **Paint(…)**, за да чертаем графичния образ на контролата.
3. Дефинираме необходимите свойства и методи.
4. Обявяваме свойствата, достъпни от дизайнера на средата за разработка (VS.NET) чрез следните атрибути:
   - **Category** – указва категорията, в която ще се показва свойството.
   - **Description** – задава описание на свойството.

Създаване на нова контрола като комбинация от други контроли

Създаването на контрола като комбинация от други контроли става по следния начин:

1. От VS.NET избираме **File | Add New Item … | UI | User Control**.
2. Използваме дизайнера на VS.NET, за да добавим контроли и да оформим желания вид на контролата.
3. Обявяваме свойствата, достъпни за дизайнера на средата за разработка чрез атрибутите **Category** и **Description**.

Създаване на нова контрола, която наследява съществуваща контрола

Създаването на нова контрола, която наследява съществуваща контрола, става по следния начин:

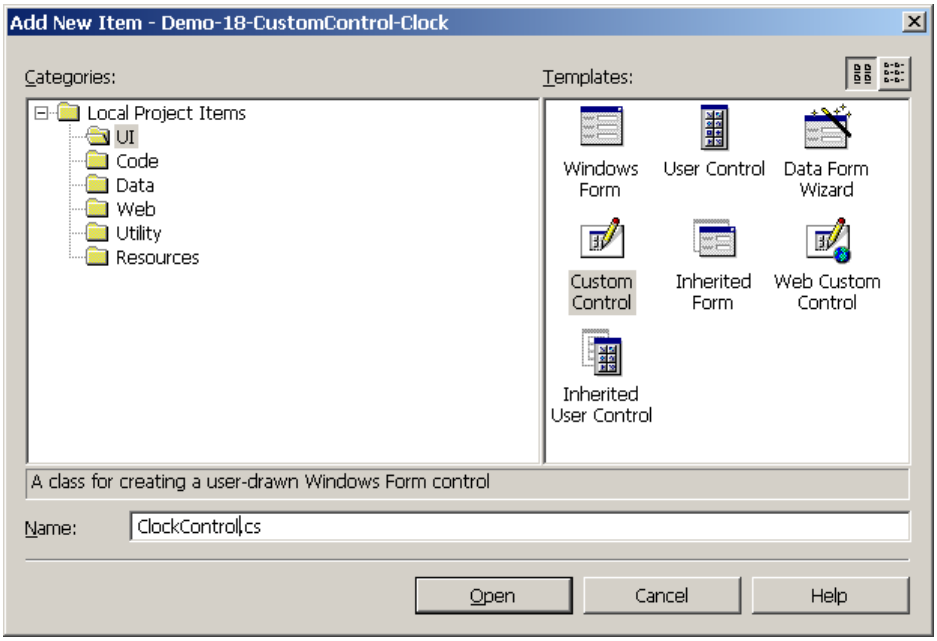
1. От VS.NET избираме **`File | Add New Item … | UI | Inherited User Control`**.
2. Избираме контролата, от която ще наследяваме.
3. Дефинираме допълнителни методи и свойства и ги обявяваме за дизайнера на VS.NET чрез атрибутите **`Category`** и **`Description`**.
4. Припокриваме **`OnXXX()`** методите при необходимост, за да променим поведението на оригиналната контрола.

Създаване на контрола – пример

В настоящия пример ще илюстрираме как със средствата на Windows Forms и GDI+ можем да създаваме потребителски Windows Forms контроли. Ще създадем контролата **`ClockControl`**, която представлява кръгъл часовник със стрелки, на който може да се задава колко часа да показва.

Ето стъпките за създаване на контролата и на приложение, което я използва:

1. Стартираме VS.NET и създаваме нов Windows Forms проект.



2. Задаваме на главната форма име **MainForm** и заглавие "**Clock Control Demo**". Променяме и името на файла от **Form1.cs** на **MainForm.cs**.
3. Създаваме нашата потребителска контрола. За целта избираме **File | Add New Item … | UI | Custom Control**. Задаваме **ClockControl** за име на контролата.
4. Дефинираме две полета **mHour** и **mMinute** и свойства за достъп до тях. Те ще съдържат часа и минутите на нашия часовник:

```
private int mHour;
private int mMinute;

[Category("Behavior"), Description("Specifies the hour.")]
public int Hour
{
   get
   {
      return mHour;
   }

   set
   {
      mHour = value;
      this.Invalidate();
   }
}

[Category("Behavior"), Description("Specifies the minutes.")]
public int Minute
{
   get
   {
      return mMinute;
   }

   set
   {
      mMinute = value;
      this.Invalidate();
   }
}
```

Приложили сме към свойствата атрибути **Category** и **Description**, за да укажем на Visual Studio .NET да ги публикува в Properties прозореца по време на дизайн. При промяна на свойствата се извиква методът **Invalidate()**, за да се пречертае контролата и да се преместят стрелките на часовника.

5. Добавяме една константа за размер по подразбиране и добавяме в конструктора код за инициализиране на контролата. Ще инициализираме контролата с текущия час:

```
private const int DEFAULT_SIZE = 100;

public ClockControl()
{
   // This call is required by the Windows.Forms Form Designer.
   InitializeComponent();

   this.Size = new Size(DEFAULT_SIZE, DEFAULT_SIZE);
   mHour = DateTime.Now.Hour;
   mMinute = DateTime.Now.Minute;
}
```

6. Припокриваме виртуалния метод **`OnPaint(…)`** и в него чертаем часовника върху **`Graphics`** повърхността на контролата. За пресмятане на координатите на стрелките използваме изчисления с помощта на тригонометрични функции синус и косинус:

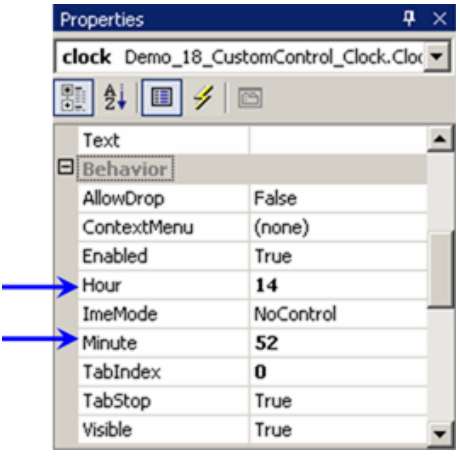
```
protected override void OnPaint(PaintEventArgs pe)
{
   Graphics g = pe.Graphics;

   // Draw the circle
   Pen pen = new Pen(Color.Blue, 1);
   g.DrawEllipse(pen, 0, 0, this.Width-1, this.Height-1);
   pen.Dispose();

   // Draw the minute finger
   double minuteFingerAngle =
      (mMinute % 60) * (2*Math.PI/60);
   int minuteFingerLen = this.Width * 45 / 100;
   int x1 = this.Width / 2;
   int y1 = this.Height / 2;
   int x2 = (int) (x1 +
     minuteFingerLen*Math.Sin(minuteFingerAngle));
   int y2 = (int) (y1 -
     minuteFingerLen*Math.Cos(minuteFingerAngle));
   pen = new Pen(Color.Red, 2);
   g.DrawLine(pen, x1, y1, x2, y2);
   pen.Dispose();

   // Draw the hour finger
   double hourFingerAngle =
      (mHour % 12) * (2*Math.PI/12) +
      (mMinute % 60) * (2*Math.PI/(60*12));
   int hourFingerLen = this.Width * 25 / 100;
   x1 = this.Width / 2;
```

```
    y1 = this.Height / 2;
    x2 = (int) (x1 + hourFingerLen*Math.Sin(hourFingerAngle));
    y2 = (int) (y1 - hourFingerLen*Math.Cos(hourFingerAngle));
    pen = new Pen(Color.Yellow, 3);
    g.DrawLine(pen, x1, y1, x2, y2);
    pen.Dispose();

    // Calling the base class OnPaint
    base.OnPaint(pe);
}
```

7. Припокриваме метода **OnSize(…)**, в който приравняваме височината и ширината на контролата и я пречертаваме. Така контролата винаги ще бъде с квадратна форма:

```
protected override void OnResize(System.EventArgs e)
{
    this.Height = this.Width;
    this.Invalidate();
}
```

8. Нашата потребителска контрола е готова. Можем да прекомпилираме приложението и да я добавим в Toolbox. За да я добавим в Toolbox, щракваме в него с десен бутон на мишката и от там избираме **Add/Remove Items**… В появилия се прозорец натискаме бутона **Browse**… и избираме изпълнимия файл на нашето приложение. Поставяме отметка пред **ClockControl** в списъка и натискаме бутона **OK**. Контролата се добавя в Toolbox.



9. В главната форма на приложението поставяме една **ClockControl** контрола с име **clock** и един панел с контроли за промяна на текущия час и минути – две **NumericUpDown** контроли с имена **NumericUpDownHour** и **NumericUpDwonMinute** и един бутон с име **ButtonSetTime** за отразяване на промените. Свойствата на

ClockControl могат да бъдат променяни от прозореца Properties (вж. фигурата по-горе).

10. Добавяме код, който при зареждане на формата (при събитие **Load** на формата) задава стойностите на **NumericUpDown** контролите за час и минута, съответстващи на тези от **ClockControl** обекта:

```
private void MainForm_Load(object sender, System.EventArgs e)
{
   NumericUpDownHour.Value = clock.Hour;
   NumericUpDownMinute.Value = clock.Minute;
}
```

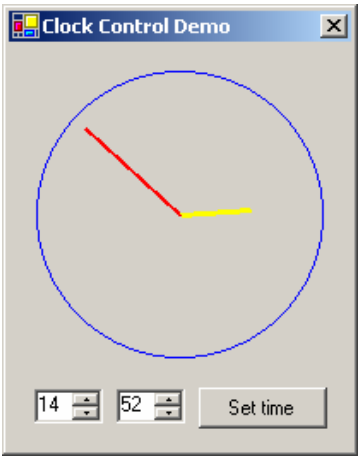
11. Добавяме обработчик на събитието **Click** на **ButtonSetTime**. В него променяме стойностите на свойствата на **ClockControl** обекта:

```
private void ButtonSetTime_Click(object sender,
   System.EventArgs e)
{
   clock.Hour = (int) NumericUpDownHour.Value;
   clock.Minute = (int) NumericUpDownMinute.Value;
}
```

12. Добавяме обработчик на събитието **SizeChanged** на формата. В него добавяме код, който не позволява на часовника да бъде върху панела:

```
private void MainForm_SizeChanged(object sender,
   System.EventArgs e)
{
   ClientSize = new Size(
      ClientSize.Width, ClientSize.Width + PanelDown.Height);
}
```

13. Приложението е готово и можем да го стартираме и тестваме.



Хостинг на контроли в Internet Explorer

Internet Explorer може да изпълнява Windows Froms контроли, вградени в тялото на HTML страници. Технологията е подобна на Java аплетите и Macromedia Flash – вгражда се изпълним код, който се изпълнява в клиентския уеб браузър. От JavaScript могат да се достъпват свойствата на Windows Forms контролите. Необходими са Internet Explorer 5.5, или по-нова версия, и инсталиран .NET Framework.

Настройките за сигурност не позволяват достъп до файловата система и други опасни действия. Сигурността може да се задава и ръчно. Ако има нужда от запазване на някакви данни на машината на потребителя, може да се използва Isolated Storage.

Хостинг на контроли в Internet Explorer – пример

Настоящият пример илюстрира как можем да реализираме хостинг на Windows Forms контроли в Internet Explorer чрез вграждането им в HTML страница и как можем да достъпваме свойствата им от JavaScript.

Да разгледаме примерна HTML страница, в която е вградена Windows Forms контролата "часовник" от предходния пример:

index.html

```
<html>

<script>

function ChangeText() {
   clockControl.Hour = hour.value;
   clockControl.Minute = minute.value;
}

</script>

<body>

   <p>Clock Control in IE</p>

   <object id="clockControl"
      classid="http:Demo-18-CustomControl-
Clock.exe#Demo_18_CustomControl_Clock.ClockControl"
      width="200" height="200">
      <param name="Hour" value="14">
      <param name="Minute" value="35">
   </object>

   <br>
   <br>
```

```
   Hour:<input type="text" id="hour"><br>
   Minute:<input type="text" id="minute"><br>
   <input type="button" value="Update the clock"
onclick="ChangeText()">

</body>

</html>
```

Как работи примерът?

Нека разгледаме по-подробно отделните части на HTML страницата. Чрез HTML тага **<object>** вмъкваме в страницата нашата контрола. Това е часовникът, който създадохме в предишния пример. Атрибутът **id**, който има стойност **clockContol**, указва идентификатор, чрез който ще можем да достъпваме обекта в HTML страницата, а атрибутите **width** и **height** указват с каква ширина и височина да се изобрази той. Атрибутът **classid** определя класа на вмъквания обект. В случая това е нашата **ClockControl** контрола. Забележете, че указваме асемблито, пространството и името на класа в стойността на този атрибут. В случая сме поставили асемблито **Demo-18-CustomControl-Clock.exe** в директорията, в която се намира и HTML страницата. Чрез таговете **<param>** задаваме стойности за свойствата на изобразяваната контрола.

Под контролата сме поставили две текстови полета и един бутон. Текстовите полета служат за въвеждане на час и минути, които да показва часовникът. Бутонът служи за промяна на стрелките на часовника. При натискането му се извиква JavaScript функцията **ChangeText()**, дефинирана в началото на страницата, която променя свойствата на контролата. Достъпът до текстовите полета и до контролата се извършва посредством техните идентификатори, зададени чрез атрибута **id**.

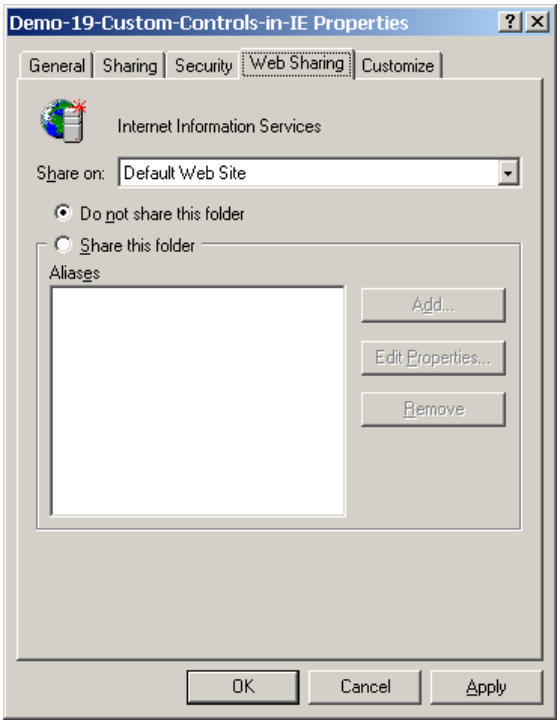
Примерът в действие

За да видим резултата от нашата работа, трябва да използваме Internet Explorer 5.5 или по-нов. Не е известен друг уеб браузър, който поддържа Windows Forms контроли.

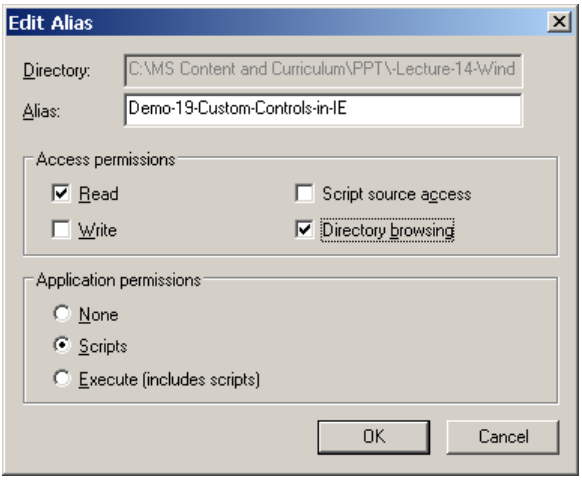
Ако отворим директно **index.html** в Internet Explorer, контролата **ClockControl** няма да се зареди заради политиката за сигурност, която не позволява локално разположени HTML документи да изпълняват Windows Forms контроли. Необходимо е страницата да бъде публикувана на някакъв уеб сървър, например IIS.

Нека файловете ни се намират в папката **Demo-19-Custom-Controls-in-IE**. Публикуването на папката в Internet Information Services (IIS) се извършва по следния начин:

1. От свойствата на папката **`Demo-19-Custom-Controls-in-IE`**, достъпни от диалоговата кутия на Windows Explorer, избираме таба "**`Web Sharing`**". В него избираме "**`Share this folder`**".



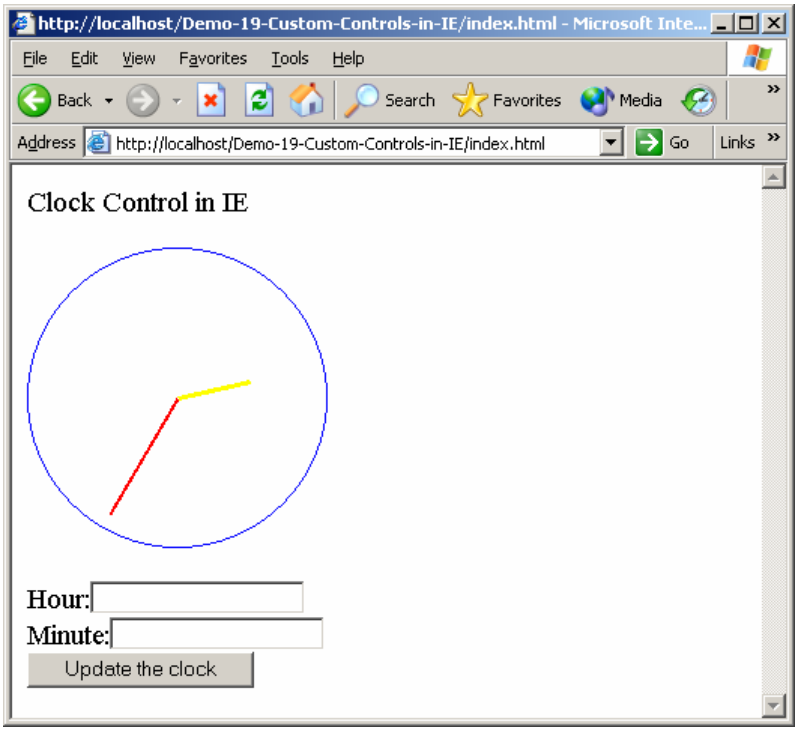
2. Публикуваме папката Internet Information Services, като позволим четене на файловете и листинг на директориите.



Сега можем да отворим с Internet Explorer URL адреса на примера от публикуваната в IIS директория:

http://localhost/Demo-19-Custom-Controls-in-IE/index.html

Ще получим следния резултат:



Ако въведем час и минута и натиснем бутона, стрелките ще променят местоположението си.

Нишки и Windows Forms

Продължителните операции в Windows Forms приложенията трябва да се изпълняват в отделна нишка. В противен случай се получава "заспиване" на потребителския интерфейс. Как можем да използваме нишки, ще разгледаме подробно в темата "Многонишково програмиране и синхронизация", но засега можем да считаме, че нишките позволяват паралелно изпълнение на програмен код в нашите приложения.

Да вземем за пример операцията "изтегляне на файл от Интернет". Тя може да отнеме от няколко секунди до няколко часа и е недопустимо приложението да блокира, докато изтеглянето на файла не приключи. В такъв случай трябва да изпълним задачата в друга нишка (thread) и от време на време да показваме на потребителя индикация за напредъка, например чрез контролата **`ProgressBar`**. Има обаче един проблем, свързан с достъпа до потребителския интерфейс при работа с нишки.

Обновяването на потребителския интерфейс на дадена контрола трябва да става само от нишката, в която работи контролата. От друга нишка безопасно могат да се извикват само методите **`Invoke()`**, **`BeginInvoke()`**, **`EndInvoke()`** и **`CreateGraphics()`**.

Никога не обновявайте Windows Forms контроли от нишка, която не ги притежава!

За изпълнение на методи от нишката, която притежава дадена контрола, използваме метода **`Invoke(…)`** на класа **`Control`**. Ето пример:

```
delegate void StringParamDelegate(string aValue);

class Form1 : System.Windows.Forms.Form
{
   private void UpdateUI(string aValue)
   {
      // Update UI here …
      // This code is called from the Form1's thread
   }

   void AsynchronousOperation()
   {
      // This runs in separate thread. Invoke UI update
      this.Invoke(new StringParamDelegate(UpdateUI),
        new object[]{"някакъв параметър"});
   }
}
```

По този начин нишката, която извършва времеотнемащата работа, работи паралелно на нишката, която управлява потребителския интерфейс, но той се обновява само от неговата нишка-собственик. Ако обновяваме потребителския интерфейс от нишката, която извършва времеотнемащата операция, а не от главната нишка на приложението, се получават много странни ефекти – от "зависване" на приложението до неочаквани изключения и системни грешки. Не го правете!

Използване на нишки в Windows Forms приложения – пример

С настоящия пример ще илюстрираме използването на нишки (threads) в Windows Forms приложения за изпълнение на времеотнемащи задачи. Ще покажем правилния начин, по който една нишка, която се изпълнява паралелно с главната нишка на Windows Forms приложението, може да обновява неговия потребителски интерфейс.

Приложението, което ще създадем, ще търси прости числа (което е време-отнемаща операция) и ще ги показва на потребителя. Търсенето ще се извършва в отделна, паралелно изпълняваща се нишка, за да не "заспива" потребителският интерфейс.

Ето стъпките за изграждане на нашето приложение:

1. Стартираме VS.NET и създаваме нов Windows Forms проект.

2. Задаваме на главната форма име **MainForm** и заглавие "**Asynchronos UI Update Demo**". Променяме и името на файла от **Form1.cs** на **MainForm.cs**.

3. Добавяме във формата два бутона с имена **ButtonStart** и **ButtonStop** и един **TextBox** с име **TextBoxLastPrimeNumber**. На свойствата **Text** на бутоните задаваме съответно стойности **Start** и **Stop**. Задаваме стойност **false** на свойството **Enabled** на бутона **ButtonStop**.

4. Добавяме променлива за нишката, която търси прости числа:

```
private Thread mPrimeNumbersFinderThread = null;
```

5. Декларираме делегат, който ще използваме при извикването на метода **Invoke(…)**, когато обновяваме потребителския интерфейс:

```
delegate void LongParameterDelegate(long aValue);
```

6. Дефинираме клас **PrimeNumberFinder**, чрез който ще търсим прости числа в интервала [0; 1 000 000 000]:

```
class PrimeNumbersFinder
{
  private MainForm mMainForm;

  public PrimeNumbersFinder(MainForm aMainForm)
  {
    mMainForm = aMainForm;
  }

  public void FindPrimeNumbers()
  {
    for (long number=0; number<1000000000; number++)
    {
      if (IsPrime(number))
      {
        mMainForm.Invoke(
        new LongParameterDelegate(mMainForm.ShowPrimeNumber),
        new object[]{number}
        );
      }
    }
  }

  private bool IsPrime(long aNumber)
  {
    // Primarity testing. Very ineffective.
    // Don't do it in a real case!!!
    for (long i=2; i<aNumber; i++)
    {
```

```
            // Just waste some CPU time
            int sum = 0;
            for (int w=0; w<100000; w++)
            {
               sum += w;
            }

            if (aNumber % i == 0)
            {
               return false;
            }
         }

         return true;
      }
   }
```

Понеже търсенето на прости числа ще се извършва в отделна нишка, в класа сме дефинирали променлива **mMainForm**, чрез която ще се обръщаме към главната форма, за да обновяваме потребителския интерфейс. Тази променлива се инициализира в конструктора на класа.

Методът **IsPrime(…)** проверява дали подаденото като параметър число е просто. Тази проверка нарочно се прави по изключително времеотнемащ, неефективен и натоварващ процесора начин, за да се симулира забавяне.

Методът **FindPrimeNumbers()** проверява последователно дали е просто всяко от числата в интервала от 0 до 1000000000. Ако числото е просто, през главната нишка на приложението се извиква методът **ShowPrimeNumber(…)**, като му се подава като параметър намереното просто число. Този метод показва числото в потребителския интерфейс. Извикването се извършва чрез метода **Invoke(…)** на формата, който има грижата да изпълни подадения му делегат през нишката, в която работи формата.

Нишката, която търси прости числа, няма право да променя директно потребителския интерфейс на приложението, защото той работи в друга нишка. Ако две нишки работят с потребителския интерфейс едновременно, могат да възникнат непредвидими проблеми – блокиране на приложението, странни изключения или странни визуални ефекти.

7. Дефинираме в главната форма метода **ShowPrimeNumber(…)**, който показва подаденото му като параметър число в текстовото поле **TextBoxLastPrimeNumber**:

```
internal void ShowPrimeNumber(long aNumber)
{
   TextBoxLastPrimeNumber.Text = aNumber.ToString();
}
```

8. Добавяме обработчик на събитието **Click** на бутона **ButtonStart**. В него деактивираме **Start** бутона, активираме бутона **Stop** и стартираме отделна нишка, в която започваме да търсим прости числа:

```
private void ButtonStart_Click(object sender,System.EventArgs e)
{
   ButtonStart.Enabled = false;
   ButtonStop.Enabled = true;
   PrimeNumbersFinder finder = new PrimeNumbersFinder(this);
   mPrimeNumbersFinderThread =
      new Thread(new ThreadStart(finder.FindPrimeNumbers));
   mPrimeNumbersFinderThread.Start();
}
```

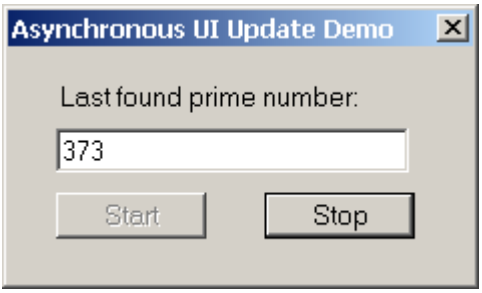
9. Добавяме обработчик на събитието **Click** на бутона **ButtonStop**. В него активираме **Start** бутона, деактивираме бутона **Stop** и прекратяваме изпълнението на стартираната нишка:

```
private void ButtonStop_Click(object sender, System.EventArgs e)
{
   ButtonStart.Enabled = true;
   ButtonStop.Enabled = false;
   mPrimeNumbersFinderThread.Abort();
}
```

10. Добавяме обработчик на събитието **Closing** на главната форма. В него прекратяваме изпълнението на нишката, търсеща прости числа (в случай че е била стартирана):

```
private void MainForm_Closing(object sender,
   System.ComponentModel.CancelEventArgs e)
{
   if (mPrimeNumbersFinderThread != null)
   {
      mPrimeNumbersFinderThread.Abort();
   }
}
```

11. Приложението е готово и можем да го стартираме и тестваме.



Въпреки че се извършва тежко изчисление и процесорът е натоварен на 100%, потребителският интерфейс не "замръзва". Ако все пак в даден момент се получи замръзване за кратко време, най-вероятно причината за това е включването на системата за почистване на паметта (Garbage Collector).

Влачене (Drag and Drop)

Реализацията на "влачене и пускане" (drag and drop) в Windows Forms приложение се извършва чрез обработването на поредица от събития.

В събитието **MouseDown** на контролата, от която започва влаченето, трябва да извикаме метода **DoDragDrop(…)**, за да копираме данните, които ще влачим.

За да дадем възможност на контрола да получава данни при влачене, трябва да зададем стойност **true** на свойството й **AllowDrop** и трябва да прихванем събитията **DragEnter** и **DragDrop**. При обработка на **DragEnter** трябва да проверяваме формата на идващите данни и да позволяваме или забраняваме получаването им. Тази проверка можем да извършим чрез метода **DragEventArgs.Data.GetDataPresent(…)**. В събитието **DragDrop** трябва да обработваме получените данни. Можем да ги извличаме посредством метода **DragEventArgs.Data.GetData(…)**.

Влачене и пускане в Windows Forms – пример

Настоящия пример илюстрира как със средствата на Windows Forms могат да бъдат реализирани приложения, които използват Drag-and-Drop технологията (влачене и пускане на обекти от една контрола към друга).

Приложението, което ще създадем, ще съдържа две контроли – едната ще се използва като източник при влаченето, а другата като получател.

Ето и стъпките за изграждане на нашето приложение:

1. Стартираме VS.NET и създаваме нов Windows Forms проект.
2. Задаваме на главната форма име **MainForm** и заглавие "**Drag and Drop Demo**". Променяме и името на файла с нейния сорс код от **Form1.cs** на **MainForm.cs**.
3. Добавяме във формата две **ListBox** контроли с имена **ListBoxSource** и **ListBoxTarget**. Те ще бъдат съответно източник и получател при влаченето.
4. Задаваме за свойството **Items** на **ListBoxSource** списък от имена на градове – **София**, **Пловдив**, **Варна**, …
5. Добавяме обработчик на събитието **MouseDown** на **ListBoxSource**. В него намираме избрания елемент от списъка и извикваме метода **DoDragDrop(…)**, с което активираме влаченето. На метода подаваме като първи параметър данните, а като втори – стойност от изброения

тип **DragDropEffects**, указваща какъв да е резултатът от влачението – в нашия случай е копиране:

```
private void ListBoxSource_MouseDown(object sender,
   System.Windows.Forms.MouseEventArgs e)
{
   Point mouseLocation = new Point(e.X, e.Y);
   int selectedIndex =
        ListBoxSource.IndexFromPoint(mouseLocation);
   if (selectedIndex != -1)
   {
      string data = (string) ListBoxSource.Items[selectedIndex];
      ListBoxSource.DoDragDrop(data, DragDropEffects.Copy);
   }
}
```

6. Задаваме на свойството **AllowDrop** на **ListBoxTarget** стойност **true**.

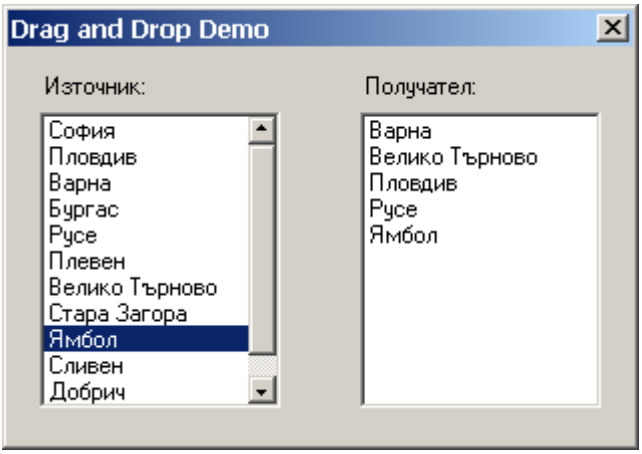
7. Добавяме обработчик на събитието **DragEnter** на **ListBoxTarget**. В него проверяваме дали влаченият обект е Unicode символен низ и съответно позволяваме или забраняваме пускането му:

```
private void ListBoxTarget_DragEnter(object sender,
   System.Windows.Forms.DragEventArgs e)
{
   if (e.Data.GetDataPresent(DataFormats.UnicodeText))
   {
      e.Effect = DragDropEffects.Copy;
   }
}
```

8. Добавяме обработчик на събитието **DragDrop** на **ListBoxTarget**. В него извличаме низа и го обработваме:

```
private void ListBoxTarget_DragDrop(object sender,
   System.Windows.Forms.DragEventArgs e)
{
   string data =
      (string) e.Data.GetData(DataFormats.UnicodeText);
   ListBoxTarget.Items.Add(data);
}
```

9. Приложението е готово и можем да го стартираме и тестваме, като завлечем няколко града от списъка-източник в списъка-получател.



Конфигурационен файл на приложението

.NET Framework приложенията могат да използват конфигурационен файл, за да четат настройките си. Той представлява обикновен XML файл:

App.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="username" value="Бай Иван" />
    <add key="language" value="US-EN" />
  </appSettings>
</configuration>
```

В тага **<appSettings>** могат да се добавят конфигурационни параметри на приложението, които представляват двойки от ключ и стойност. Настройките от конфигурационния файл могат да бъдат извличани по време на изпълнение по следния начин:

```
string username = System.Configuration.
  ConfigurationSettings.AppSettings["username"];
// username = "Бай Иван"
```

От VS.NET можем да добавим конфигурационен файл като изберем **File | Add New Item… | Application configuration file | App.config**. При компилация **App.config** се копира под име **<име_на_проекта.exe.config>**.

Извличане на настройки от конфигурационен файл – пример

Настоящият пример илюстрира как можем да извличаме настройки от конфигурационния файл на приложението. Ще създадем приложение, което извлича стойност от своя конфигурационен файл и я показва.

Ето и стъпките на изграждане на нашето приложение:

1. Стартираме VS.NET и създаваме нов Windows Forms проект.
2. Задаваме на главната форма име **MainForm** и заглавие "**Config File Demo**". Променяме и името на файла от **Form1.cs** на **MainForm.cs**.
3. Добавяме във формата един **TextBox** с име **TextBoxUserName** и един бутон с име **ButtonReadUserName**. Задаваме на свойството **Text** на бутона стойност "**Read user name from config file**".
4. Добавяме конфигурационен файл на приложението, като избираме **File | Add New Item… | Application configuration file | App.config**. В него добавяме нов конфигурационен параметър с ключ **username** и стойност "**Бай Иван**":

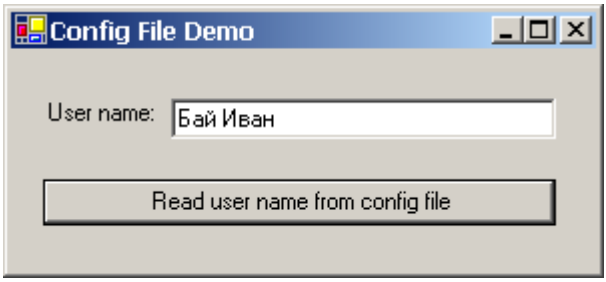
App.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="username" value="Бай Иван" />
  </appSettings>
</configuration>
```

5. Добавяме обработчик на събитието **Click** на **ButtonReadUserName**. В него извличаме стойността на параметъра **username** и я показваме в текстовото поле:

```
private void ButtonReadUserName_Click(object sender,
  System.EventArgs e)
{
  TextBoxUserName.Text = System.Configuration.
    ConfigurationSettings.AppSettings["username"];
}
```

6. Приложението е готово и можем да го стартираме и тестваме:



Упражнения

1. Какво представлява библиотеката Windows Forms? Каква функционалност предоставя? Кога се използва?
2. Какво е компонент? Какво представлява компонентният модел .NET Framework? Какво е характерно за него?
3. Опишете програмния модел на Windows Forms. Каква функционалност реализира той?
4. Кои са най-важните класове от Windows Forms? Кои са най-важните им методи и свойства?
5. Какво е характерно за всички Windows Forms контроли? Кои са общите им методи и свойства?
6. Какво е характерно за формите в Windows Forms? Какви свойства и събития имат те?
7. Как се поставят контроли в дадена форма? Как се прихващат събития, породени от дадена контрола?
8. Реализирайте Windows Forms приложение, което представлява опростен вариант на стандартния калкулатор в Windows. Калкулаторът трябва да поддържа основните аритметични операции с цели и реални числа.
9. Със средствата на Windows Forms реализирайте играта "Хвани бягащия бутон". Играта представлява една форма, в която има един бутон със заглавие "Натисни ме". При приближаване на курсора на мишката в близост до бутона той трябва да "бяга от него" (да се премества на друго място във формата, възможно по-далече от курсора на мишката).
10. Със средствата на Windows Forms реализирайте проста информационна система за управление на клиентите на дадена фирма. Системата трябва да визуализира списък от клиенти (**`ListBox`**) и да позволява добавяне, редактиране и изтриване на клиенти. Всеки клиент е или юридическо или физическо лице. Юридическите лица се описват с наименование, вид (ЕТ, АД, ООД, сдружение, ...), Булстат, данъчен номер, адрес, телефон, email, уеб сайт и МОЛ (който е физическо лице). Физическите лица се описват с име, презиме, фамилия, пол, ЕГН, данни за лична карта, адрес, телефон и email. Приложението трябва да се състои от 3 форми – главна форма, съдържаща клиентите, форма за въвеждане/редакция на юридическо лице и форма за въвеждане/редакция на физическо лице. Използвайте подходящи Windows Forms контроли във формите. Данните трябва да се четат и записват в XML файл.
11. Със средствата на Windows Forms реализирайте специализиран редактор за библиотеки с текстови документи. Една библиотека представлява съвкупност от текстови документи, организирани дървовидно в

папки. В една папка може да има документи и други папки (подобно на файловата система на Windows). Всеки документ представлява някакъв текст с форматиране. Редакторът трябва да може да създава библиотеки, да чете/записва библиотеки от/в XML файл. Когато е отворена дадена библиотека, редакторът трябва да позволява редактиране на документите в нея (промяна на текста и форматирането на отделни фрагменти от него), както и създаване/изтриване/преименуване на папки и документи. За дървото с папките трябва да се използва контролата **`TreeView`**, а за активния документ - **`RichEdit`**. Редакторът трябва да разполага с падащо меню, 2 контекстни менюта (за дървото с папките и за полето за редактиране на документ), 3 ленти с инструменти (за отваряне/записване на библиотека, за работа с дървото с папките и за форматиране на активния в момента документ), статус лента и подходящи кратки клавиши за по-важните команди. Реализирайте и търсене и заменяне на текст в документите.

12. Напишете Windows Forms приложение, в което се въвежда информация за физическо лице (име, презиме, фамилия, ЕГН, адрес, телефон, email, личен сайт) и въведеното се записва в XML файл. Реализирайте валидация на всяко едно от полетата и на цялата форма, като използвате подходящи регулярни изрази.

13. Със средствата на Windows Forms и простото свързване на данни (simple data binding) реализирайте приложение за управление на проста система с информация за градове и държави. Всяка държава се описва с име, език, население, национален флаг и списък от градове. Всеки град се описва с име, население и държава. Трябва да се реализира навигация по градовете и държавите и редакция на информацията за тях, като не се използват списъчни контроли, а само текстови полета и просто свързване. Да се реализира четене и записване на данните в XML файл.

14. Със средствата на Windows Forms и сложното свързване на данни (complex data binding) реализирайте система, подобна на системата за управление на информация за градове и държави. Добавете към системата списък от континенти за всяка държава. За визуализацията и навигацията използвайте таблици (**`DataGrid`**) и списъчни контроли. Реализирайте предходното приложение, като съхранявате данните не в XML файл, а в релационна база от данни (напр. MS SQL Server). Използвайте разкачения модел за достъп до данните (disconnected model), като реализирате възможност за разрешаване на конфликтите, които възникват при работа с много потребители едновременно.

15. Създайте Windows Forms приложение, с което могат да се въвеждат данни за физически и юридически лица. Физическите лица се описват с име, ЕГН, адрес, телефон, email и уеб сайт. Юридическите лица се описват с наименование, вид (ЕТ, АД, ООД, сдружение, ...), Булстат, данъчен номер, адрес, телефон, email, уеб сайт и МОЛ (име и ЕГН на физическо лице). Използвайте наследяване на форми, като отделите в базова форма общите елементи на потребителския интерфейс и

общите полета от формите за въвеждане на физически и юридически лица.

16. Реализирайте Windows Forms приложение, което по ежедневните данни от дадено техническо измерване за даден период (текстов файл с цели положителни числа) визуализира графично резултатите като редица от правоъгълни стълбове. При обемни данни осигурете възможност за скролиране на графиката.

17. Със средствата на Windows Forms реализирайте играта "морски шах" (в квадратна дъска с размери 3 на 3 се поставят пулове "X" и "0"). Играчът трябва да може да играе срещу компютъра в 2 режима: "компютърът играе оптимално" и "компютърът играе хаотично (случайно)". Осигурете подходяща визуализация и интерактивност на играта.

18. Реализирайте Windows Forms MDI приложение, което може да отваря файлове с графични изображения (gif, jpg, png) и може да ги преоразмерява и да ги записва в друг файл.

19. Реализирайте Windows Forms приложение, което показва даден текстов файл, като визуализира всеки негов ред със специален ефект: всяка буква първоначално се появява на случайно място във формата и започва да се придвижва анимирано към мястото си. За 2 секунди всяка буква трябва да си е на мястото. След изчакване от 1 секунда се преминава към следващия ред от входния файл.

20. Със средствата на Windows Forms реализирайте прост текстов редактор, който може да отваря файлове с влачене от Windows Explorer.

21. Наследете контролата **`TextBox`** и създайте потребителска контрола **`NumberTextBox`**, която позволява въвеждане само на числа.

22. Направете Windows Forms потребителска контрола **`HourMinuteBox`**, която се състои от 2 **`NumericUpDown`** полета и позволява въвеждане на час и минута в интервала [0:00 - 23:59].

23. Реализирайте Windows Forms потребителска контрола "зарче", която представлява квадрат, в който могат да се изобразяват графично стойности от 1 до 6 (както са при стандартните зарчета при някои игри). Контролата трябва да реализира собствено изчертаване и свойство "**`Value`**" за задаване на текущата стойност.

24. С помощта на контролата "зарче" реализирайте играта "състезание": Двама играчи играят последователно. При всеки ход играчът, който е на ход, хвърля 2 зарчета (генерират се случайни стойности) и мести толкова стъпки, колкото е сумата от хвърлените зарове. Печели първият, който премине сумата 50. Реализирайте подходяща визуализация на позицията на двамата играчи на хвърлените зарове.

25. Реализирайте играта "състезание" като Windows Forms контрола и я хостнете в Internet Explorer, използвайки подходяща уеб страничка. Хвърлянето на заровете извиквайте с JavaScript при натискане на бутон от уеб страницата.

26. Със средствата на Windows Forms реализирайте приложение, което търси текст във всички файлове в дадена директория. Понеже търсенето е бавна операция, реализирайте я в отделна нишка. При намиране на текста добавяйте файла и отместването, на което е намерен, в **`ListBox`** контрола чрез главната нишка на приложението, като използвате **`Invoke()`** метода на формата. Реализирайте възможност за прекратяване на търсенето. Реализирайте подходяща визуализация при щракване върху някое от намерените съвпадения в резултата.

27. Реализирайте Windows Forms приложение, което съдържа една текстова контрола, стойността на която се зарежда от конфигурационния XML файл на приложението. При изход от приложението стойността на тази контрола трябва да се запазва обратно в конфигурационния файл. За четене от конфигурационния файл използвайте **`System.Configuration.ConfigurationSettings.AppSettings`**, а за писане в него използвайте DOM парсера на .NET Framework.

Използвана литература

1. Светлин Наков, Графичен потребителски интерфейс с Windows Forms – http://www.nakov.com/dotnet/lectures/Lecture-14-Windows-Forms-v1.0.ppt

2. MSDN Library - http://msdn.microsoft.com

3. Microsoft Windows Forms QuickStarts Tutorial – http://www.csharpfriends.com/quickstart/winforms/doc/default.aspx

4. Marj Rempel, Kenneth S. Lind, Marjorie Rempel, MCAD/MCSD Visual C# .NET Certification All-in-One Exam Guide, McGraw-Hill, 2002, ISBN 0072224436

5. MSDN Library, Event Handling in Windows Forms – http://msdn.microsoft.com/library/en-us/vbcon/html/vbconeventhandling.asp

6. Threading in Windows Forms – http://www.yoda.arachsys.com/csharp/threads/winforms.shtml

7. J. Fosler, Windows Forms Painting: Best Practices – http://www.martnet.com/~jfosler/articles/WindowsFormsPainting.htm

Глава 16. Изграждане на уеб приложения с ASP.NET

Автори

Михаил Стойнов
Рослан Борисов
Стефан Добрев
Деян Варчев
Иван Митев
Христо Дешев

Необходими знания

- Базови познания за езика C#
- Базови познания за архитектурата на .NET Framework
- Базови познания по Интернет технологии
- HTTP (Hyper Text Transfer Protocol)
- HTML (Hyper Text Markup Language)
- Познания за архитектурата на уеб базираните приложения

Съдържание

В тази тема ...

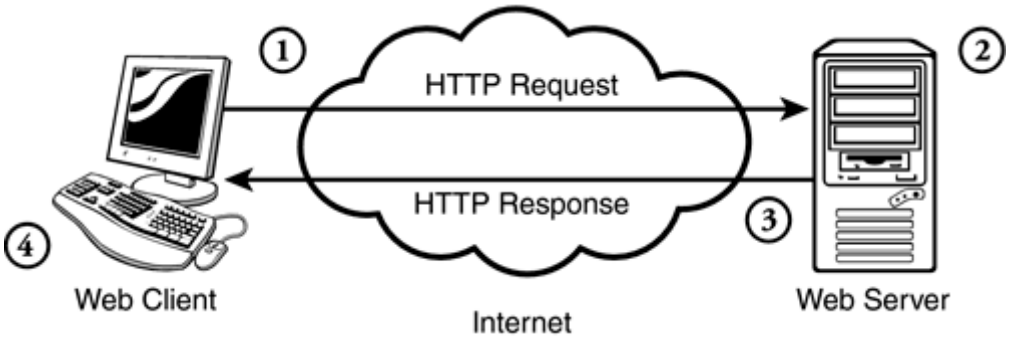
В настоящата тема ще разгледаме разработката на уеб приложения с ASP.NET. В началото ще запознаем читателя с уеб формите и техните основни директиви, атрибути и тагове. Ще разгледаме видовете уеб контроли, които се използват при изграждане на уеб приложения, и по-важните от тях. Ще разгледаме концепцията за отделяне на кода от потребителския интерфейс (code-behind), ще обясним програмния модел на ASP.NET и работата със събития. След това ще демонстрираме как да работим с данни, извлечени от релационна база от данни. Ще обърнем специално внимание на принципите на свързване на контроли с данни (data binding) и ще обясним как да свързваме списъчни и итериращи контроли. Ще разгледаме как можем да управляваме вътрешното състояние на уеб приложението: работа със сесии и cookies, достъп до контекста на приложението и технологията ViewState. Ще покажем как да валидираме данни, въведени от потребителя, чрез различните валидатори. Ще обясним концепцията за потребителските контроли като метод за преизползване на части от приложението. Ще се научим как да проследяваме и дебъгваме уеб приложения. Ще покажем как се оптимизират, конфигурират и разгръщат ASP.NET уеб приложения (кеширане, настройки и deployment). Ще обърнем специално внимание и на сигурността при уеб приложенията.

Въведение

ASP.NET е библиотека за разработка на уеб приложения и уеб услуги, стандартна част от .NET Framework. Тя дава програмен модел и съвкупност от технологии, чрез които можем да изграждаме сложни уеб приложения.

Изпълнение на ASP.NET уеб приложение

Уеб приложенията използват модела заявка-отговор (request-response), както е показано на фигурата:



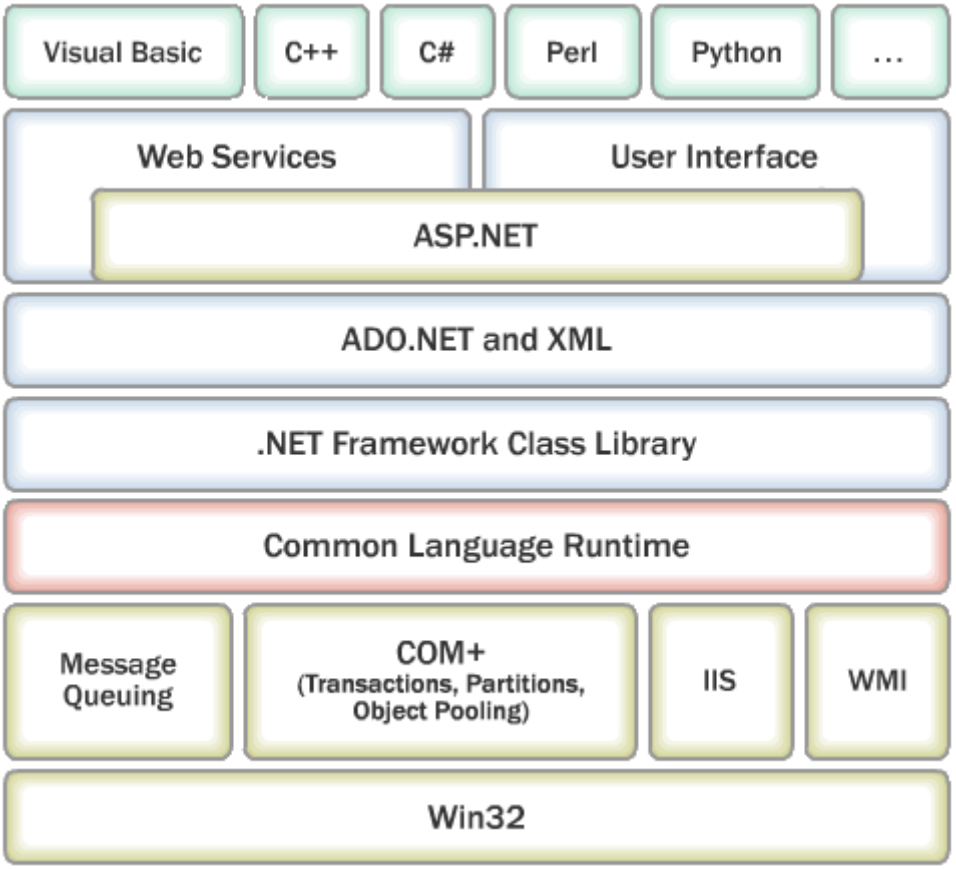
1. Потребителят въвежда в браузъра адрес на страница (URL). Браузърът изпраща HTTP заявка (request) към уеб сървъра.
2. Сървърът получава заявката и я обработва. В случая с ASP.NET, IIS намира процес, който може да обработи дадената заявка.
3. Резултатът от вече обработената заявка се изпраща обратно към потребителя/клиента под формата на HTTP отговор (response).
4. Браузърът показва получения отговор като уеб страница.

Преглед на технологията ASP.NET

ASP.NET е програмна платформа за разработка на уеб приложения, предоставена от .NET Framework. Тя предлага съвкупност от класове, които работят съвместно, за да обслужват HTTP заявки. Също като класическите ASP (Active Server Pages), ASP.NET се изпълнява на уеб сървър и предоставя възможност за разработка на интерактивни, динамични, персонализирани уеб сайтове, както и на уеб базирани приложения. ASP.NET е също и платформа за разработка и използване на уеб услуги.

ASP.NET и .NET Framewrok

На фигурата са показани основните компоненти на .NET Framework, част от които е библиотеката ASP.NET.

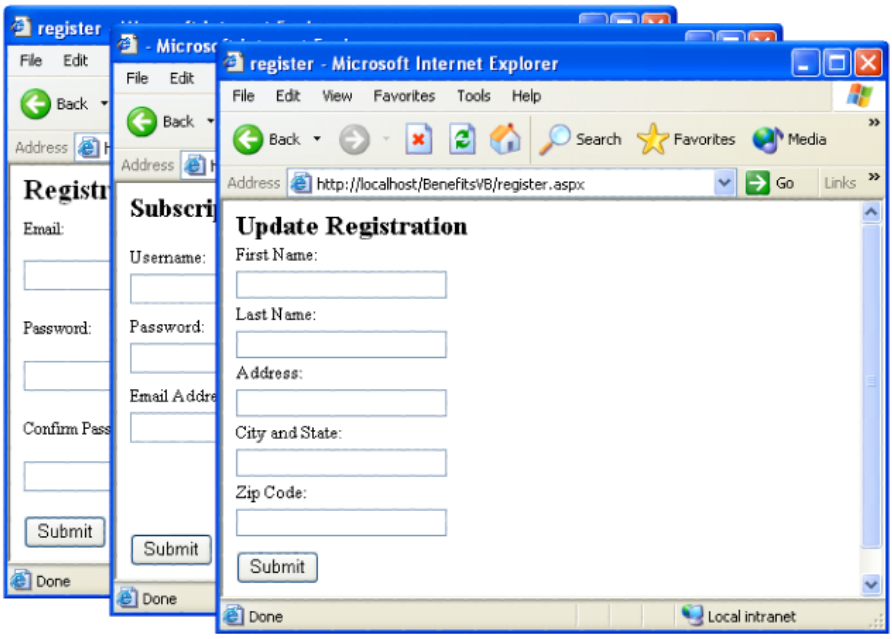


Разлики между ASP и ASP.NET

Разликите между ASP и ASP.NET са значителни. ASP.NET предлага ново ниво на абстракция за разработка на уеб приложения. Ключова характеристика на ASP.NET е възможността за разделяне на кода описващ дизайна от кода, реализиращ логиката на приложенията. ASP.NET приложенията могат да бъдат разработвани с помощта на всички езици за програмиране, които се компилират до MSIL код (C#, VB.NET, J#, Managed C++ и много други).

Фундаменти на ASP.NET

Основният компонент на ASP.NET е уеб формата – абстракция на HTML страницата, която интернет потребителите виждат в браузъра си. Замисълът на създателите на ASP.NET е работата с уеб формите да бъде интуитивна и максимално улеснена, както е при Windows Forms формите. ASP.NET предлага едно високо ниво на абстракция, предоставяйки ни богат избор от уеб контроли, подобни на тези в Windows Forms, и намалява нуждата програмиста да работи с чист HTML код.

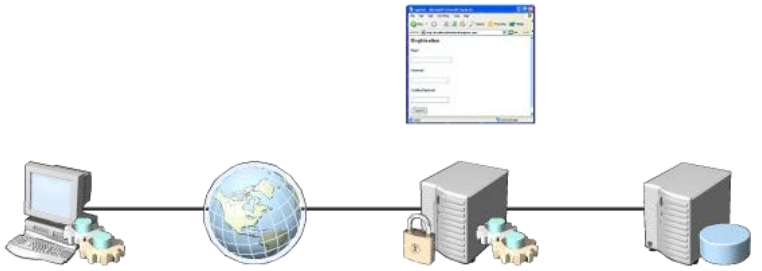


Всяко ASP.NET приложение се изгражда от една или повече уеб форми, които могат да взаимодействат помежду си, създавайки интерактивна система.

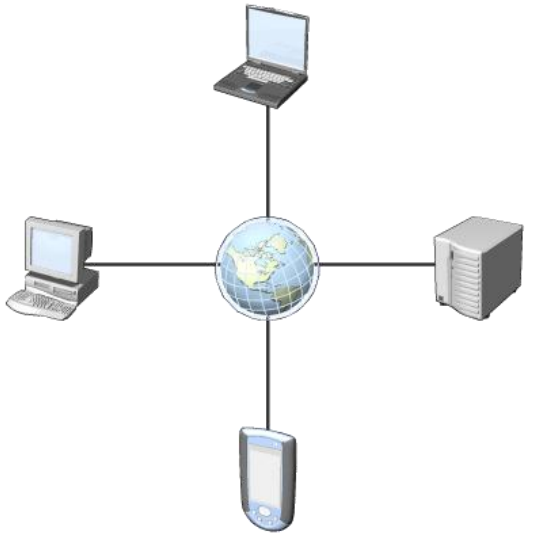
Как работи ASP.NET?

Традиционните уеб страници могат да изпълняват код на клиента, с който извършват сравнително прости операции.

ASP.NET уеб формите могат да изпълняват и код от страна на сървъра (server-side code). С него те генерират HTML код, който да се върне като отговор на заявката. За целта могат да се извършват обработки, изискващи достъп до бази от данни и до ресурсите на самия сървър, генериращи допълнителни уеб форми и други.



Всяка уеб форма в крайна сметка се трансформира в HTML код, пригоден за типа на клиентския браузър. Това позволява улеснена разработка на уеб форми. Те работят практически върху всяко устройство, което разполага с интернет свързаност и уеб браузър.



Разделяне на визуализация от бизнес логика

Един от основните проблеми на класическите ASP беше смесването на HTML с бизнес логика. Това правеше страницата трудна за разбиране, поддръжка и дебъгване. Файловете ставаха големи и сложни и се забавяше процеса на разработка на приложението. Една от основните архитектурни цели на ASP.NET е справянето с този проблем. Тъй като реализацията на потребителския интерфейс и на бизнес логика са до голяма степен, две независими задачи, ASP.NET предоставя модел за разработка, при който те са физически разделени в отделни файлове.

Програмирането за клиентския интерфейс (UI) се разделя на две части:

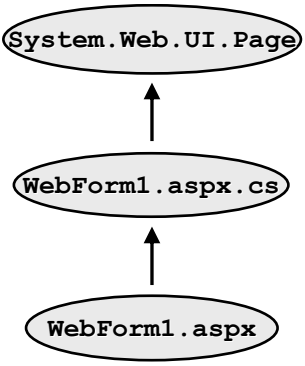
- За визуализация се използва HTML-подобен код, записан във файл с разширение **`.aspx`**.

- Бизнес логиката се дефинира в отделен файл (с разширение **.cs** за C# или **.vb** за Visual Basic .NET), съдържащ конкретната имплементация на определен програмен език.

Файлът, съдържащ бизнес логиката, се нарича "Изпълним код на уеб формата" (Code-behind).

Зад всяка уеб форма стои богатият обектен модел на .NET Framework и тя се компилира до клас в асемблито на проекта ни.

Класът, генериран от **.aspx** файл, се непряк наследник на **Page** класа. Съществува междинен клас в йерархията, който е за изпълнимия код (code-behind class). В него можем лесно да добавяме методи, обработка на събития и др.



Чрез "изпълнимия код" представянето е разделено от логиката. Това улеснява значително поддръжката на **.aspx** страниците.

Компоненти на ASP.NET

Ето кои са основните компоненти, от които се изграждат уеб приложенията, базирани на ASP.NET:

Web Forms – описват интерфейса за ASP.NET приложение.

Code-behind класове – асоциират се с уеб форми и контроли и съдържат server-side код.

Web.config – файл, съдържащ конфигурацията на ASP.NET приложението.

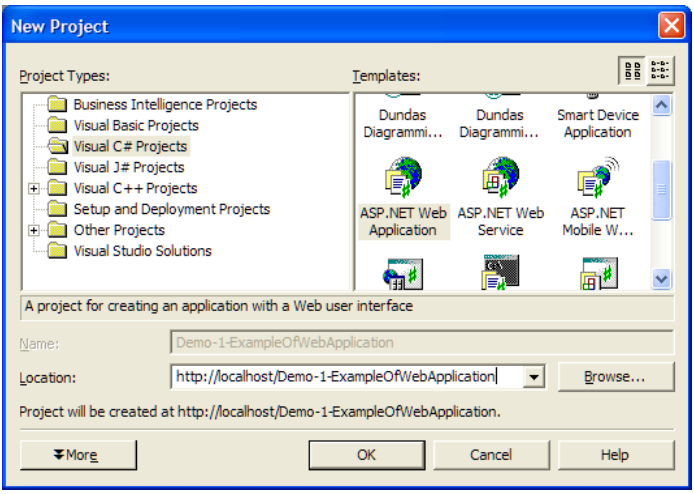
Machine.config – файл с глобални настройки за уеб сървъра.

Global.asax – файл, съдържащ код за прихващане на събития на ниво приложение (application level events).

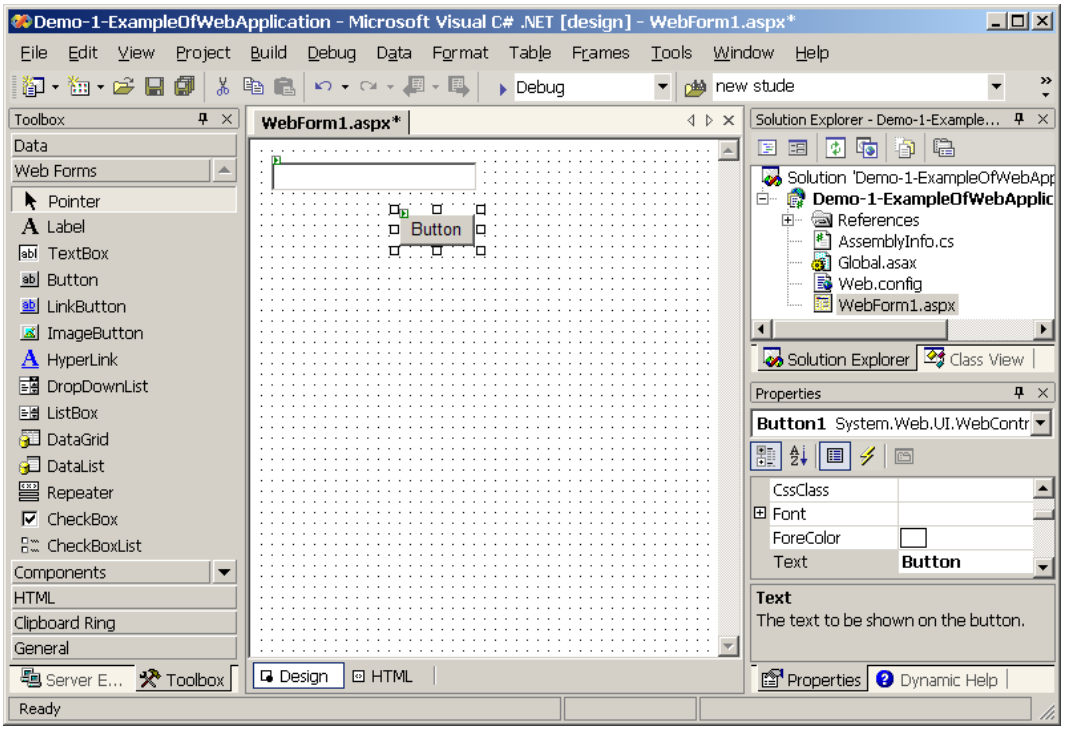
Съществуват и други компоненти като **Http Modules**, **Http Handlers** и други, но на тях няма да се спираме.

Пример за уеб приложение

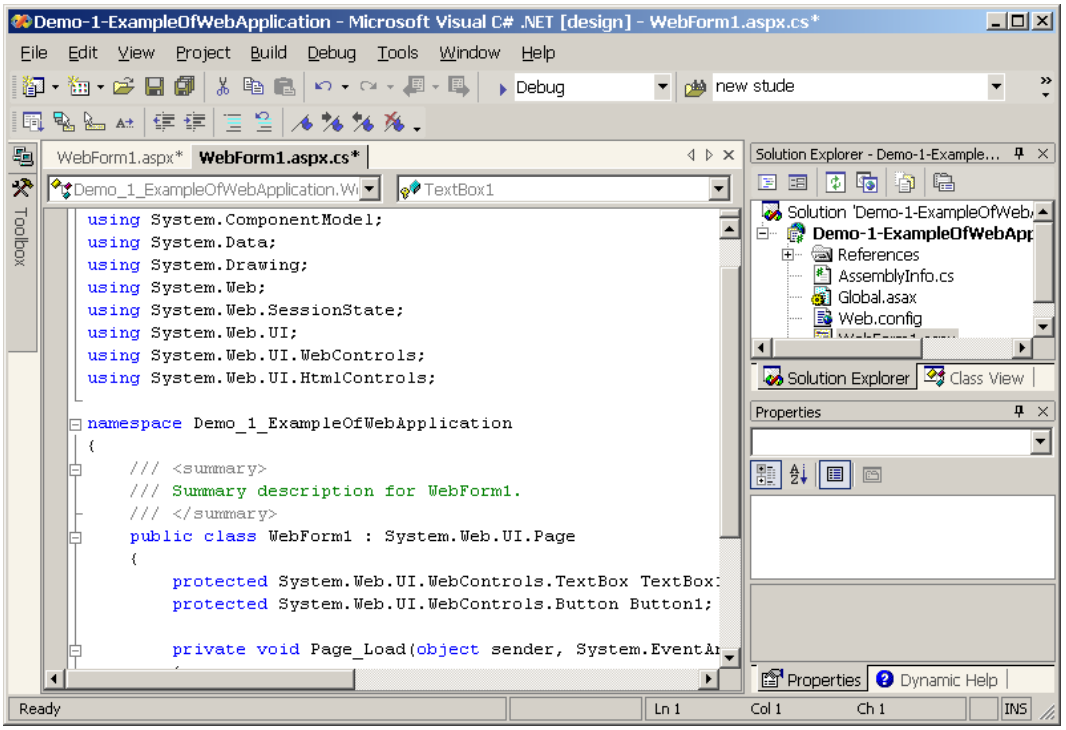
Ще създадем примерно уеб приложение чрез Visual Studio.NET, за да разгледаме структурата на директориите и файловете му. Отваряме Visual Studio.NET и създаваме нов уеб проект с име **`Demo-1-ExampleOfWebApplication`**:



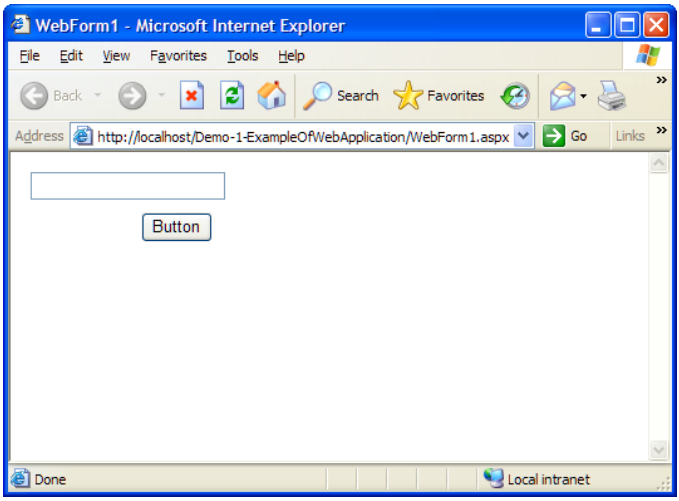
Отляво е кутията с инструменти (toolbox), където са контролите. Нека добавим текстово поле и бутон, като привлачим двете контроли върху формата.



Можем да разгледаме кода на страницата (code-behind класа) като върху формата натиснем [F7].



Можем да компилираме и стартираме приложението с [Ctrl+F5].



ASP.NET Web Application проекти във VS.NET

При създаване на проект за .NET уеб приложение (ASP.NET Web Application) във Visual Studio .NET, се създават две директории.

В едната (по подразбиране това е **\My Documents\Visual Studio Projects**) се намира solution файла (**.sln**). Той описва проектите, участващи в приложението. Обикновено в поддиректории се съхраняват останалите проекти от solution файла.

Втората директория е с идентично име и се създава в уеб-достъпна папка, като най-често това е папката **c:\inetpub\wwwroot**. Тя съдържа файловете, които са нужни на уеб приложението: **.aspx** страниците, техните code-behind файлове и файловете **Web.config** и **Global.asax**. Когато се компилира проекта, се създава съответно асембли в директорията **bin**.

Забележка: Директорията **c:\inetpub\wwwroot** е коренната (root) виртуална директория по подразбиране на уеб сървъра Internet Information Server (IIS), с който по подразбиране работи Visual Studio.NET за разгръщане на приложенията си.

Виртуална директория е такава, която се вижда през протокола HTTP, например:

Ако отваряме сървъра локално, можем да го извикаме с адреса **http://localhost/**.

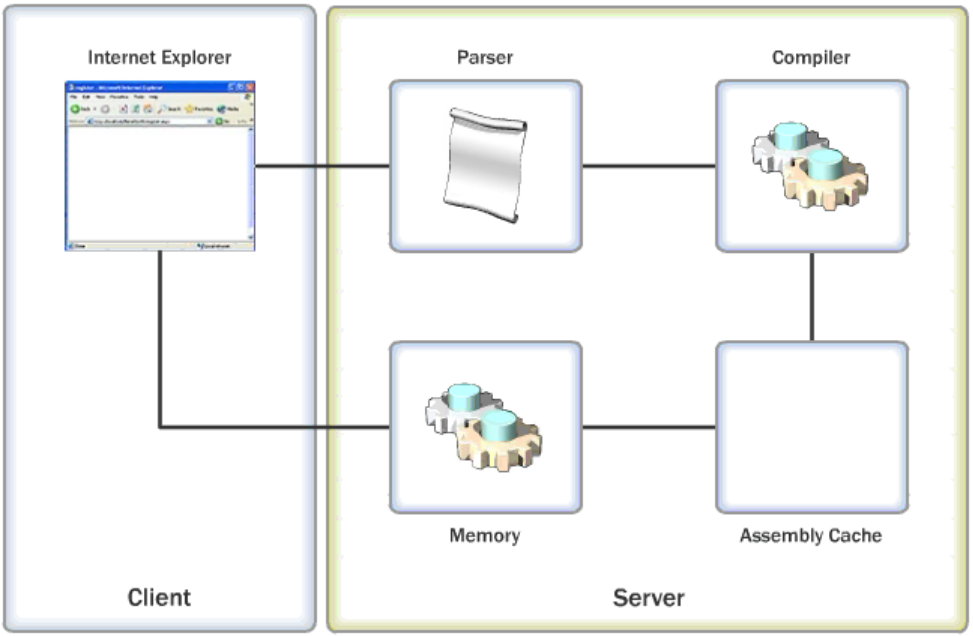
Като направим директорията **c:\inetpub\wwwroot\Lecture-15-ASP.NET-and-Web-Applications\Demo-1-ExampleOfWebApplication**, тя ще се вижда през протокола HTTP като виртуалната директория:

http://localhost/Lecture-15-ASP.NET-and-Web-Applications/Demo-1-ExampleOfWebApplication/.

Модел на изпълнение на ASP.NET

Моделът на изпълнение (execution model) на ASP.NET е следният:

1. Клиентският браузър изпраща HTTP GET заявка на сървъра.
2. Сървърът намира за коя страница е заявката и започва да я изпълнява.
3. ASP.NET парсерът интерпретира нейния сорс код.
4. Ако кодът не е бил компилиран в асембли, ASP.NET извиква компилатора.
5. Средата за изпълнение (CLR) зарежда в паметта и изпълнява Microsoft Intermediate Language (MSIL) кода.
6. Страницата се изпълнява и генерира HTML код. Сървърът връща този резултат на клиента като HTTP отговор.



Уеб форми

ASP.NET уеб приложенията представляват най-общо съвкупност от уеб форми. Нека разгледаме как можем да създаваме и използваме уеб форми.

Какво е уеб форма (Web Form)?

Уеб формата е програмируема уеб страница. Тя служи като потребителски интерфейс в ASP.NET уеб приложенията. Уеб формите съдържат HTML код и контроли. Те се изпълняват на уеб сървъра. Най-често това е Microsoft Internet Information Server (IIS). Уеб формите връщат като резултат потребителски интерфейс, под формата на HTML код, който се изпраща на клиентския браузър.

Създаване на уеб форма

Функциите на уеб формата се дефинират, като се използват три нива на атрибути.

Атрибутите на **`@Page`** директивата дефинират глобална функционалност. Атрибутите на **`body`** тага дефинират как ще се покаже една страница. Атрибутите на **`form`** тага дефинират как ще се обработят групите контроли.

```
<%@ Page Language="c#"
   Codebehind="WebForm1.aspx.cs"
   Inherits="WebApplication1.WebForm1"%>
<html>
```

```
  <head><title>WebForm1</title></head>
  <body MS_POSITIONING="GridLayout">
    <form id="Form1" method="post">
      <asp:Button ...></aspButton>
    </form>
  </body>
</html>
```

Забележка: **@Page** директивата е специална конструкция, използвана в ASP.NET уеб формите. Въпреки че и в HTML има **<body>** и **<form>** тагове, същите (когато са записани така **<body runat="server" ...>** и **<form runat="server" ...>**) играят по-специална роля в ASP.NET. Тези особености са обяснени в детайли по-нататък.

Поддържат се два вида разполагане на HTML елементите на страницата.

- **FlowLayout**: HTML обектите се нагласят по ширината на прозореца на браузъра.
- **GridLayout**: HTML обектите са с абсолютни координати на HTML страницата. Това е стойността по подразбиране.

Директиви

Директивите предоставят възможност за контролиране на компилацията и изпълнението на уеб формата. Името на всяка директива започва с "@" и е заградено с **<%** и **%>** тагове. Директивите могат да бъдат поставени навсякъде в **.aspx** файла на формата, но обикновено се поставят в началото му. Настройките и опциите към всяка директива се задават като атрибути.

Важни директиви:

- **@Page** – главна директива за формата (по-късно разгледана);
- **@Import** – въвежда дадено пространство от имена във формата;
- **@Implements** – указва, че формата (или контролата) имплементира даден интерфейс;
- **@Control** – аналог на **@Page** директивата (използва се само за потребителски контроли);
- **@OutputCache** – контролира способността за кеширане на формите;
- **@Register** – регистрира контрола за употреба в уеб форма;
- **@Reference** – декларативно указва, че сорс файлът на друга потребителска контрола или форма трябва да бъде динамично компилиран и свързан с формата, в която е декларирана директивата.

Ето един пример за използване на **@Page** директивата:

```
<%@ Page Language="c#" Codebehind="WebForm1.aspx.cs"
Inherits="WebApplication1.WebForm1"%>
```

Директивата <@Page ...>

Дефинира специфични за формата (**.aspx** файл) атрибути, използвани от парсера и компилатора на ASP.NET.

Важни атрибути:

- **AutoEventWireup** – решава автоматичното абониране за събитията на страницата и контролите.
- **Culture** – определя културата (регионалните настройки), която да се използва при обработка на данните.
- **UICulture** – определя културата за видимите от потребителя текстови съобщения.
- **Debug** – определя дали тази страница е компилирана с дебъг символи (debug symbols).
- **EnableSessionState** – определя дали ще се поддържа сесията.
- **EnableViewState** – определя дали ще се използва "view state".
- **ErrorPage** – определя страница, към която ще се пренасочва в случай на необработено изключение.

Атрибути на директивата <@Page ...>

Чрез използването на **@Page** атрибути се дефинират глобални свойства на уеб формата. Тагът **<@ Page>** дефинира специфични за страницата атрибути. Те се използват от парсера за страници на ASP.NET и компилатора. В даден **.aspx** файл може да бъде включи само един **<@ Page>** таг.

Атрибутът **Language** дефинира програмния език на скрипта в уеб страницата. Най-често използвани са: Visual Basic.NET и C#, като се поддържат и Visual J#, JScript.NET и т.н.

Атрибутът **CodeBehind** сочи към code-behind страницата (файла), който съдържа логиката на уеб формата. При създавана на уеб форма във Visual Studio.NET (например **WebForm1.aspx**), се създава автоматично и code-behind клас във файл с име: **WebForm1.aspx.vb** или **WebForm1.aspx.cs**.

Атрибутът **SmartNavigation** в ASP.NET инструктира браузъра да опреснява само тези секции от формата, които са се променили. Технологията Smart Navigation премахва премигването на екрана при опресняване. Scroll позицията се запазва и "last page" в историята остава същата. Smart Navigation е достъпен само за ползватели на Microsoft Internet Explorer 5 или по-нов.

Тагът <form>

Тагът <form> дефинира как ще бъдат обработени контролите. Той е различен от едноименния таг в езика HTML – тук дефинира контейнер на контроли за цялата уеб страница. На една уеб форма може да има много **<form>** HTML елементи, но само един от тях може да е сървърна контрола в **.aspx** страницата.

HTML страница	ASP.NET страница (само една форма)
<form>…</form> <form>…</form> <form>…</form>	<form runat="server">…</form> <form>…</form> <form>…</form>

Вградени обекти в ASP.NET

Всяка страница в ASP.NET приложението е наследник на класа **Page**, който предлага множество полезни свойства. Сега ще изброим по-важните, а по-късно ще разгледаме в детайли начина на употреба на повечето от тях:

- **Application** (клас **HttpApplication**) – приложение;
- **Session** (клас **HttpSession**) – сесия;
- **Request** (клас **HttpRequest**) – заявка;
- **Response** (клас **HttpResponse**) – отговор;
- **Server** (клас **HttpServerUtility**) – сървър;
- **Context** (клас **HttpContex**) – контекст;
- **Cache** (клас **System.Web.Caching.Cache**) – кеш.

Ако искаме да препратим изпълнението към друга страница, можем да използваме два метода: чрез свойствата **Response** и **Server**.

- **Response.Redirect("Login.aspx")** – пренасочване от страна на клиента (client redirection). Променя адреса на уеб браузъра.
- **Server.Forward("WebForm1.aspx")** – пренасочване от страна на сървъра (server redirection). Запазва адреса на уеб браузъра. На практика Web браузърът не разбира за пренасочването.

Уеб контроли

Уеб контролите представляват компоненти, от които се изграждат ASP.NET уеб формите. Те се изпълняват на сървъра по време на зареждане на уеб формата и се "рендират" (трансформират) до HTML контроли, които след това се визуализират от клиентския уеб браузър.

ASP.NET сървърни контроли

ASP.NET сървърната контрола (ASP.NET server control) е компонент, който се изпълнява на сървъра и обвива потребителски интерфейс и друга функционалност. Сървърните контроли се използват в ASP.NET страниците и code-behind класовете. Сред тях има контроли за бутони, текстови полета (text boxes), падащи (drop-down) списъци и други.

Всички server контроли имат атрибутите **id** и **text**, както и **runat="server"** атрибута. Последният атрибут означава, че логиката (кода) на контролата се изпълнява на сървъра, а не при клиента, както е с HTML елементите.

Сървърните контроли ни дават възможност за обработка на събития в код, изпълняван на сървъра. Обработчик на събитие е процедура, която се изпълнява в резултат потребителско действие (щракане на бутон, списък и др). Кодът, който се изпълнява, се слага в така наречените събитийни (event) процедури. Вие като програмист на уеб форми решавате как да обработвате събитията на съответната контрола.

Общ модел на ASP.NET сървърните контроли

В ASP.NET сървърните контролите са базирани на общ модел и като резултат споделят голям брой атрибути.

Например, когато трябва да се смени цветът на фона на контрола, винаги се използва едно и също свойството **BackColor**. Следният HTML код, който описва server контрола, показва някои типични атрибути:

```
<asp:Button id="Button2" runat="server" BackColor="red"
Width="238px" Height="25px" Text="Web control"></asp:Button>
```

Специфичен за различните браузъри HTML код

Когато една страница се подготвя за клиента (rendering), сървърната контрола доставя HTML код, съобразен с вида на браузъра, подал заявката.

Например, ако браузърът поддържа възможност за четене на скрипт (client-side scripting), какъвто е Internet Explorer version 4.0 или по-нова, контролите създават такъв клиентски скрипт (client-side script), за да си имплементират част от функционалността директно на клиентския браузър. Ако не поддържа клиентски скрипт, контролите създават код, изпълняван на сървъра (server-side код), който имплементира същата функционалност, но за да се получи същата функционалност се правят повече заявки до сървъра.

Следният код е отрязък от ASP.NET уеб форма, която ще създаде текстово поле с текст "Enter your Username":

```
<asp:TextBox id="Textbox2" runat="server" Width="238px"
Height="25px">Enter your Username</asp:TextBox>
```

Когато тази страница се достъпва от потребител с Internet Explorer 6, средата за управление (CLR) създава следния HTML код:

```
<input name="TextBox1" type="text" value="Enter your Username"
id="TextBox1" style="height:25px;width:238px" />
```

Server контролите могат да генерират различен HTML код за отделните браузъри. Примерно контролът **Panel**, води до генериране на ** ** в Internet Explorer 6, а на другите браузъри **<table>**. За разработчиците това става прозрачно и ни позволява да се концентрираме върху логиката на приложението.

HTML сървърни контроли (HTML server controls)

HTML сървърните контроли наподобяват традиционните HTML елементи.

HTML елементите се третират от уеб формата като обикновен текст и техните атрибути не са достъпни за сървъра. С конвертирането на HTML елементи към HTML server контроли е възможен достъпа до техните атрибути от server-side код. Това конвертиране позволява контролите да инициират събития, които се обработват на сървъра.

HTML server контролите имитират HTML елементите синтактично, с разликата, че включват атрибута **runat="server"**. За тях има изискване да са вложени в контейнер **<form "runat="server">...</form>**. Намират се в пространството от имена **System.Web.UI.HtmlControls**.

HTML сървърни контроли – пример

Ще направим уеб приложение, в което ще има проста HTML страница. След това стъпка по стъпка от нея ще създадем уеб форма с HTML контроли в нея. За база ще използваме HTML таговете от първоначалната страница. Нека имаме следния код в HTML страницата:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
  <head>
    <title>Test HTML page</title>
  </head>
  <body>
    <form method="post">
      Username: <br />
      <input type="text" name="username" /><br />
      Password: <br />
      <input type="text" name="password" /><br />
      <input type="submit" name="submit" value="submit" />
    </form>
  </body>
</html>
```

Това е обикновена HTML форма за потребителско име и парола.

Първото изискване, за да направим тази HTML страница уеб форма, е да добавим **runat="server"** в HTML таговете за въвеждане на данни (**<input type="text" …>**), както и на обграждащата ги HTML форма (**<form …>**).

Това е достатъчно, за да компилираме страницата като ASP.NET форма.

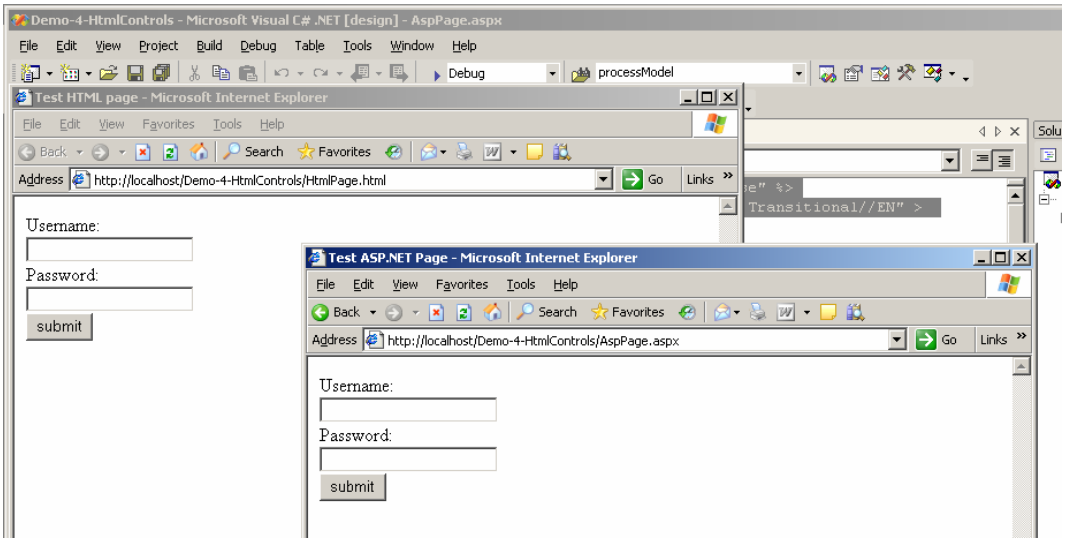
Ако искаме да добавяме код директно в страницата, а не в code-behind клас трябва да добавим директивата **<%@Page ... %>** като минимум укажем програмния език, който ще използваме (C#, Visual Basic.NET, Jscript.NET …).

Ако искаме да достъпваме контролите (**<input type="text" runat="server" …>**) от code-behind класа, трябва да им добавим и име, с което да бъдат достъпвани **id="UsernameTextbox"**. Променлива със същото име трябва да бъде налична и в code-behind страницата.

След всички тези промени ето как изглежда нашата нова ASP.NET страница:

```
<%@ Page language="C#" AutoEventWireup="false" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >

<html>
  <head>
    <title>Test ASP.NET Page</title>
  </head>
  <body>
    <form id="LoginForm" method="post" runat="server">
     Username: <br/>
     <input type="text" id="TextboxUsername" runat="server"
name="Username"/><br />
     Password: <br />
     <input type="text" id="TextboxPassword" runat="server"
name="Password"/><br />
     <input type="submit" id="submit" value="submit"
runat="server" name="submit"/>
    </form>
  </body>
</html>
```

Когато компилираме и стартираме новата ни уеб форма, тя изглежда и работи така, както очакваме:



Кодът на HTML страницата не се е променил, докато ето какво получаваме като отворим сорс кода на уеб формата в браузъра (View → Source в Internet Explorer):

File Edit Format View Help

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >

<html>
        <head>
                <title>Test ASP.NET Page</title>
        </head>
        <body>
    <form name="LoginForm" method="post" action="AspPage.aspx" id="LoginForm">
<input type="hidden" name="__VIEWSTATE"
value="dDwtMTM2MzcONjE5Nzs7Pvswwe4PKO6AQ+911UxlmgxNah2Q" />

                Username: <br/>
                <input name="Username" id="Username" type="text" /><br />
                Password: <br/>
                <input name="Password" id="Password" type="text" /><br />
                <input name="submit" id="submit" type="submit" value="submit"
/>
    </form>
  </body>
</html>
```

Кодът на уеб формата е близък до този на HTML страницата, когато формата е компилирана.

Уеб сървърни контроли (Web server controls)

Web server контролите са сървърни контроли, създадени специално за ASP.NET. Те включват не само form-type контроли като бутони и текстови кутии, но също контроли със специално предназначение като календар контролата (виж фигурата по-долу). Web server контролите са по-абстрактни от HTML server контролите. Техният обектен модел не е задължително да наподобява синтаксиса на HTML.

<			Май 2005 г.			>
понеделник	вторник	сряда	четвъртък	петък	събота	неделя
25	26	27	28	29	30	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	1	2	3	4	5

Характеристики на уеб сървърните контроли

Web server контролите се базират на общ модел и всички споделят общи характеристики: тага **<asp:ControlType...>** и атрибута **id**. Web server контролите няма да функционират без атрибута **runat="server"**. Web server контролите се намират в пространството от имена **System.Web.UI.WebControls** и могат да се използват във всяка уеб форма.

```
<asp:Button id="MyBtn" runat="server">
<asp:Calendar id="MyCal" runat="server">
<asp:TextBox id="MyTxt" runat="server">
```

Кои контроли да ползваме?

Когато се създават ASP.NET страници, има възможност да се използват HTML server контроли или Web server контроли. Могат да се смесват безпрепятствено. Например - за бързо преправяне на HTML страница в ASP.NET страница.

Използването на Web server контролите е препоръчително пред HTML server контролите, тъй като те имат повече възможности и по-богат обектен модел.

Използвайте HTML сървърни контроли, ако:

- Предпочитате HTML-подобен обектен модел. HTML server контролите имат почти същия синтаксис като HTML елементите. HTML server контролите имат server-side функционалност също като Web server контролите.
- Работите със съществуващи HTML страници и искате бързо да им добавите функционалността на уеб формите. Заради това, че HTML server контролите съответстват точно на HTML елементите, не е нужно да подменяте контроли и да рискувате да се появят грешки при форматирането на страницата заради подмяната.
- Контролите трябва да изпълняват едновременно client-side и server-side скрипт. Можете да пишете client-side скрипт и да посочвате контролите с HTML името им (**<tag name="html_name">**), защото при

клиента те са HTML елементи. В същото време можете да пишете server-side код, защото те са server контроли.

- Скоростта на връзката (bandwidth) е ограничена и трябва да обработвате голяма част от данните при клиента, за да намалите натовареността.

Използвайте Web server контроли, ако:

- Предпочитате компонентния модел на .NET. Ще имате възможността да използвате обектно-ориентирано програмиране, ще можете да идентифицирате контролите по техния **`id`** атрибут и лесно да разделяте логиката от потребителския интерфейс. С Web server контролите ще можете да създавате приложения с вложени контроли и да управлявате събитията (events) на ниво контейнер на контроли (container level events).
- Създавате уеб страници, които ще бъдат разглеждани на различни браузъри. Контролите има способността да генерират HTML, който е съобразен с възможностите на клиентския браузър и се възползва от всички негови предимства. Така можете да пишете за последните версии на браузърите, без да се притеснявате, че по-старите версии няма да могат да работят с пълната функционалност на страницата ви.
- Имате нужда от специфична функционалност като например календар или банери, която е имплементирана само в Web server контроли. В Интернет има огромно разнообразие от Web server контроли, които решават специфичен проблем.
- Скоростта на връзката (bandwidth) ви не е твърде ограничена и увеличен брой връщания до сървъра (цикли от заявка-отговор) от клиента няма да създадат проблеми със скоростта.

Категории уеб сървърни контроли

Web сървър контролите се делят на:

- Вътрешните контроли (**Intrinsic controls**) съответстват на прости HTML елементи като бутони или списъци. Използват се по същия начин като HTML server контролите.
- Контролите за валидация (**Validation controls**) включват логика, която позволява валидация на потребителски въведени данни по унифициран начин. Контролата за валидация трябва да се прикрепи към контрола, приемаща потребителски вход и да се опишат правилата за валиден вход.
- Обогатените контроли (**Rich controls**) включват по-сложни функции. Пример е **`AdRotator`** контролата, която се използва за показване на последователност от картинки (използва се за банери) или **`Calendar`** контролата, която представлява календар.

- **List-bound** контролите могат динамично да показват данни на ASP.NET уеб страница. Дават възможността за показване, форматиране на изхода, сортиране и промяна.
- **Internet Explorer** уеб контролите са група сложни контроли, например **`MultiPage`**, **`TabStrip`**, **`Toolbar`**, and **`TreeView`** контролите, които могат да бъдат свалени от Интернет и да се интегрират във Visual Studio .NET за използване във всяко ASP.NET уеб приложение. Тези контроли могат да бъдат изобразени като стандартен HTML, но могат да се възползват и от допълнителните възможности на Internet Explorer 5.5 или следващи версии, при което имат по-богата функционалност.

Вътрешни контроли и съответствие с HTML

Вътрешните (intrinsic) Web server контроли отговарят на прости HTML елементи. Някои от често използваните вътрешни Web server контроли са показани в таблицата:

Вътрешни уеб контроли	HTML тагове
`<asp:button>`	`<input type="submit">`
`<asp:checkbox>`	`<input type="checkbox">`
`<asp:hyperlink>`	`<a href="..."></a>`
`<asp:image>`	`<img src="...">`
`<asp:imagebutton>`	`<input type="image">`
`<asp:linkButton>`	
`<asp:label>`	`<span> </span>`
`<asp:listbox>`	`<select size="5"></select>`
`<asp:panel>`	`<div> </div>`
`<asp:radiobutton>`	`<input type="radio">`
`<asp:table>`	`<table> </table>`
`<asp:textbox>`	`<input type="text">`

Контроли за валидация

Контролите за валидация са скрити контроли, които проверяват данните, въведени от потребителя, срещу предефинирани правила. Ще изброим набързо някои от често използваните контроли за валидация

- **`RequiredFieldValidator`** – изисква входът да е стойност, различна от празната (т.е. да се въведени някакви данни).
- **`CompareValidator`** – проверява дали стойността на контролата е равна, по-голяма или по-малка от друга.

- **RangeValidator** – изисква входът да е в някакви граници (обхват).
- **RegularExpressionValidator** – изисква входът да отговаря на предефиниран регулярен израз (например пощенски код, телефонен номер …).
- **CustomValidator** – позволява задаването на произволно условие, което може да се дефинира и изпълни и на клиента и на сървъра (например условие за просто число).
- **ValidationSummary** – събирателна контрола, която може да извежда съобщенията за грешка на всички контроли за валидация.

Обогатените контроли (Rich controls)

Обогатените контроли (Rich controls) са специфични контроли, които решават сложна, но често срещана задача. Техни представители са:

- **AdRotator** – показва последователност (предефинирана или случайно генерирана) от изображения. Най-често се използва за банери.
- **Calendar** – показва графично представяне на интерактивен календар.

Списъчни контроли (List-bound controls)

List-bound контролите могат да показват данни от източник (най-често бази от данни). Някои от най-често използваните са описани по-долу:

- **CheckBoxList** – показва данните като колона от check boxes.
- **DropDownList** – показва данни като падащ списък.
- **Listbox** – показва списък от елементи в кутийка.
- **RadioButtonList** – показва данните като колона от бутони за алтернативен избор (radio buttons).
- **Repeater** – показва информация (от **DataSet** или масив), като повтаря потребителски дефиниран шаблон. Шаблонът описва как се представя всеки един елемент. В този шаблон най-често има други контроли.
- **DataList** – подобна на контролата **Repeater**, но с повече форматиращи и layout опции, включително и възможността данните да се показват в таблица. **DataList** контролата също позволява да се определи и поведение при редактиране на данните.
- **DataGrid** – показва данните в табличен вид. Доставя механизми за редактиране, сортиране и страниране.

Code-behind

Отделянето на програмния (C#) код, свързан с презентационната логика на приложението, от потребителския му интерфейс значително улеснява поддръжката на уеб приложенията.

В ASP.NET кодът на **aspx** страниците обикновено се отделя от програмния (C#) код, който ги управлява. Този програмен код се грижи за подготовката на страницата за визуализация и за взаимодействието с потребителя и е известен още като "презентационна логика". В него се обработват събитията, предизвикани от контролите в уеб формата.

За отделянето на презентационната логика от потребителския интерфейс обикновено с всяка **aspx** страница е свързан един C# клас – файл с разширение **aspx.cs**. Този файл е известен като "code behind" и се поддържа автоматично от VS.NET.

Добавяне на код в уеб форма

Добавянето на код в ASP.NET уеб форма ви дава възможност да предоставите функционалността, от която потребителят се нуждае. Без код вашето уеб приложение може да изглежда добре, но няма да прави нищо.

Добавянето на код в уеб форма става по един от три начина:

- **Mixed code** – кодът е в същия файл, в който е и уеб съдържанието. Този метод не се препоръчва, защото води до сложен и труден за поддържане код. Използвал се е при ASP приложенията. Такъв метод видяхме в някои от примерите от последната демонстрация.
- **Inline code** – кодът е отделен в отделна SCRIPT секция в същия файл.
- **Code-behind** – кодът е в code-behind страница – отделен файл от този на HTML съдържанието. Когато използвате Visual Studio.NET, това е методът по подразбиране.

Inline code

Когато се използва inline код, HTML кодът и inline кодът са в отделни секции на един и същ **.aspx** файл. Това разделение е за яснота, когато се чете страницата. Двете секции могат да се намират навсякъде по страницата.

```
<html>
  <asp:Button id="btn" runat="server"/>
...
</html>

<script Language="c#" runat="server">
  private void btn_Click(object sender, System.EventArgs e)
```

```
  {
    ...
  }
</script>
```

Code-behind класове

Code-behind класовете представляват отделни компилирани класове, който съдържат програмната логика на страницата. Всяка уеб страница в едно уеб приложение има собствена code-behind страница. По подразбиране code-behind страницата има същото име като уеб страницата, с която е асоциирана. Разширението на файла е **.aspx.vb** или **.aspx.cs** в зависимост от това какъв език е бил използван. Когато уеб приложението се изпълнява двата файла формират цялата страница.

Как работи code-behind?

За да асоциира една **.aspx** страница с нейната code-behind страница, Visual Studio .NET добавя три атрибута към **@Page** директивата:

- **Inherits** – позволява на **.aspx** страницата да наследява code-behind класа.
- **Codebehind** – използва се вътрешно от Visual Studio .NET, за да асоциира файловете.
- **Src** – съдържа името на code-behind страницата. Използва се, ако уеб приложението не е прекомпилирано.

```
<%@ Page Language="c#"
  Inherits="MyProject.WebForm1"
  Codebehind="WebForm1.aspx.cs"
  Src="WebForm1.aspx.cs" %>
```

JIT компилация

Code-behind страницата може или да бъде прекомпилирана от Visual Studio .NET, когато се компилира уеб приложение, или да бъде just-in-time (JIT) компилирана при първата заявка.

Ако **Src** атрибутът липсва от **<@Page ...>** директивата в **.aspx** файла, при build на приложението страницата се компилира. По подразбиране Visual Studio .NET не добавя атрибута **Src**, т.е. всички code-behind страници са компилирани, когато се стартира приложението. Прекомпилирането спестява забавянето при първа заявка за съответната страница. Друго предимство е, че няма нужда сорс кодът на code-behind страниците да се разпространява до уеб сървъра.

Когато се използва JIT компилация, code-behind страницата се компилира при първа заявка. Съответно първата заявка е по-бавна.

Събития

Когато потребител взаимодейства с уеб форма (щрака, избира, въвежда данни), се генерира събитие (event). Действието, което трябва да се извършва в отговор, се реализира в събитийна (event) процедура.

Прихващане на събития

Нека създадем контрола, за да генерираме събитие в уеб формата. Visual Studio .NET декларира променлива в code-behind страницата с име като **id** атрибута на контролата.

Следният HTML код дефинира уеб форма и бутон с **id="Button1"**:

```
<form id="Form2" method="post" runat="server">
     <asp:Button id="Button1" runat="server"/>
</form>
```

В code-behind страницата се декларира променлива със същото име:

```
protected System.Web.UI.WebControls.Button Button1;
```

Когато щракнем с мишката два пъти върху този бутон, VS.NET прихваща събитието "натискане на бутона" и генерира метод, който се извиква при неговото настъпване. Ето как изглежда генерираният метод:

```
private void Cmd1_Click(object sender, System.EventArgs e)
{
   // Event handling goes here ...
}
```

Самото абониране за събитието на бутона става по начина, по който става в Windows Forms. VS.NET добавя следния код в инициализационната част на страницата:

```
this.Button1.Click +=
   new System.EventHandler(this.Button1_Click);
```

Свойството AutoEventWireup

Свойството AutoEventWireup указва дали събитията автоматично да се връзват към страницата.

Ако е **true**, следният код е излишен (изпълнява се автоматично):

```
this.Init += new System.EventHandler(this.Page_Init);
this.Load += new System.EventHandler(this.Page_Load);
```

ASP.NET сам намира методи с имена като **Button1_Click** и ги извиква като обработчик на събитието **Click** за контрол **Button1**.

При нужда от висока производителност, се препоръчва да не се използва автоматично връзване.

Жизнен цикъл на ASP.NET страниците

При всяка заявка за ASP.NET страница серия от събития се случват в строго определена последователност, известна като "жизнен цикъл на страницата" (page event life cycle). Тук ще изброим и обясним някои от по-важните събития:

Page_Init – служи за инициализиране на Web server контролите в страницата. По време на изпълнението му не трябва да се осъществява достъп до контролите. Може да се използва за заделяне на ресурси.

Page_Load – извиква се всеки път, когато страницата бъде поискана - както при първоначално отваряне на страницата, така и след потребителско действие (примерно натискане на бутон в нея)**.** Събитието се използва за извличане на данните, попълнени в контролите, както и за промяна на състоянието на контролите в страницата.

Събитията на контролите служат за обработки в code-behind частта, които са свързани с отделните контроли от страницата (например **Button1_Click**, **TextBox1_Changed**).

Page_PreRender – извиква се преди да се започне рендирането (rendering) на страницата. Рендирането на страницата е процесът на създаване на изходния HTML код от **.aspx** страницата, включващ контролите в нея и данните, получени от потребителя до момента. На тази стъпка стойностите на контролите са възстановени от визуалното състояние (view state) и могат да бъдат нанесени последни промени, които да бъдат записани обратно в него, преди страницата да се покаже в браузъра.

Page_Unload – извиква се при приключване на рендирането на страницата. Използва се за освобождаване на ресурси.

Свойството IsPostBack

В ASP.NET предназначението на формите е да връщат информация обратно към сървъра за обработка. Този процес се нарича "postback". Със свойството **IsPostBack** на класа **Page** може да се провери дали страница се зарежда за пръв път. Ако **IsPostBack** е **true**, трябва да се изпълни първоначалния инициализационен код, а ако е **false** (т.е заявката е предизвикана от контрола на страницата), да се изпълни код, който отговаря на събитието, предизвикало връщането на страницата.

Ако страницата се инициализира с код, който трябва да се изпълни само веднъж (например попълване на DataSet от

	базата данни), винаги проверявайте дали тя се отваря за първи път или е в резултат на postback!

Ето един пример:

```
private void Page_Load(object sender,
   System.EventArgs e)
{
   if (!Page.IsPostBack)
   {
      // Executes only on initial page load
      // Initialize controls here
   }

   // This executes on every request
   // Controls are already initialized
}
```

Свойството AutoPostBack

Ако искате новата стойност на контролата да бъде незабавно изпратена на сървъра, без да чакате потребителят да натисне на някой бутон, можете да укажете стойност **true** на свойството **AutoPostBack** на контролата. В момента, в който потребителят промени стойността на контролата, информацията ще се изпрати на сървъра. Сървърът осъвременява контролите на страницата и ги връща на клиента. Така страницата става по-гъвкава и интерактивна.

В следващия примерен HTML код **ListBox** контролата използва свойството **AutoPostBack**. Всеки път, когато потребителят променя стойността ѝ, страницата се праща на сървъра автоматично и възниква събитието **SelectedIndexChanged** (т.е., ако има метод, асоцииран с него, той ще се изпълни):

```
<asp:DropDownList id="ListBox1" runat="server"
     AutoPostBack="True">
   <asp:ListItem>First Choice</asp:ListItem>
   <asp:ListItem>Second Choice</asp:ListItem>
</asp:DropDownList>
```

Добавяме и код в code-behind страницата, който да покаже новоизбраната стойност в текстово поле:

```
private void ListBox1_SelectedIndexChanged
   (object sender, System.EventArgs e)
{
   TextBox1.Text = ListBox1.SelectedItem.Value;
}
```

HTML escaping проблеми

Когато получаваме данни от клиент, трябва винаги да проверяваме дали специалните знаци в текстовите променливи са описани правилно.

В HTML езика, знаците за по-малко и по-голямо ('<' и '>') са специални символи и браузърите интерпретират думата между тях като оператор или команда. Понякога се налага да използваме тези знаци като част от текст. За да визуализираме такъв знак, трябва да използваме специално записване (escaping). Тази техника се нарича escaping, защото често се реализира с поставяне на обратно наклонена черта ('\') преди знака.

Например знаците за по-малко и по-голямо ('<' и '>') се записват така: **`<`** и **`>`** (lt = less than, gt = greater than).

В HTML записването на един или повече интервала винаги се възприема като един интервал и визуално се представя като един интервал. Ако искате да поставите повече от един интервал, използвате специалния знак **` `** (nbsp = nonbreaking space).

За повече информация погледнете "The HTML Coded Character Set" (http://www.w3.org/MarkUp/html-spec/html-spec_13.html) в сайта на World Wide Web Consortium (W3C) или хипервръзките на края на тази глава.

Правилата за кодирането на специалните знаци създават потенциални проблеми при получаването на текстови данни от потребителя, които след това трябва да се изпишат в HTML страница.

HTML escaping проблеми – пример

Ето един прост пример, в който възниква escaping проблем:

```
string userName = "\"<script language='JavaScript'>while(1)
alert('bug!')</script>";
```

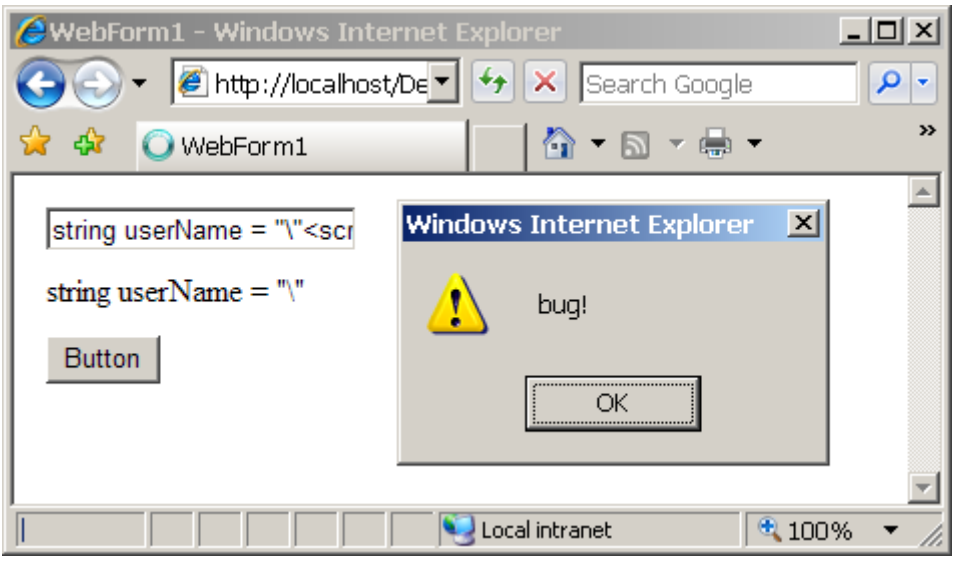
Потребителят е изпратил горния символен низ. Ако го присвоим на **`TextBox`** няма да има проблеми:

```
TextBox1.Text = userName;
```

Ако обаче го присвоим на етикет, съдържанието му се интерпретира като HTML код и този код ще бъде изпълнен:

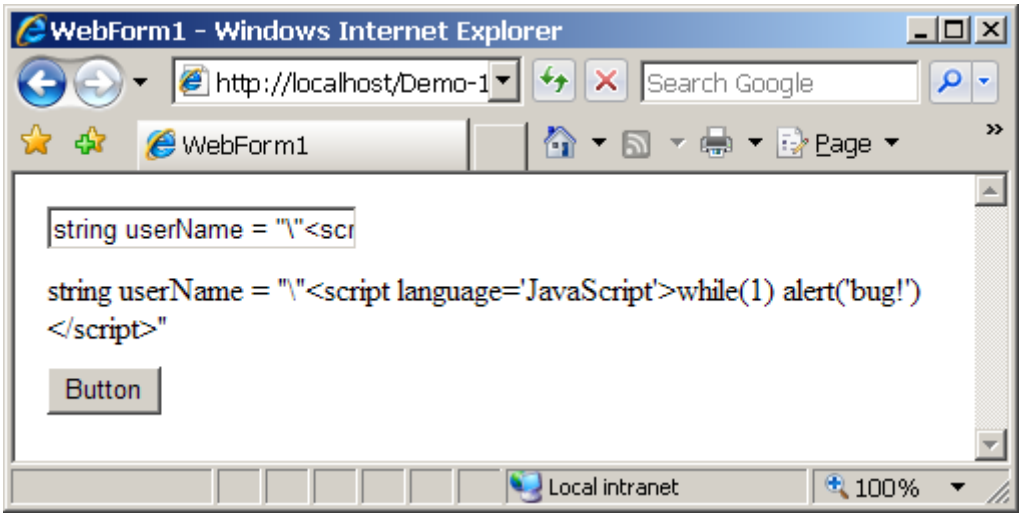
```
Label1.Text = userName;
```

Вместо да се изпише като текст в етикета, в браузъра се появява следното съобщение:



Решението на този проблем е да се използва статичният метод **`Server.HtmlEncode(…)`**:

```
Label1.Text = Server.HtmlEncode(userName);
```



Сега всичко е както трябва. Ако разгледаме как е записан кодът в изходната HTML страница, ще забележим, че е била променена само първата кавичка:

```
<input name="InputTextBox" type="text" value="\&quot;<script
language='JavaScript'>while(1) alert('bug!');</script>" …>
```

Свързване с данни (Data binding)

ASP.NET предлага нов декларативен синтаксис за свързване с данни (data binding). Този изключително гъвкав синтаксис позволява свързването не само с бази от данни, но и със свойства, колекции, изрази, дори резултати от методи. В HTML-подобния код на уеб формите свързването на данните става в секции от вида **<%# %>**. Ето няколко примера:

- Със свойство:

```
Customer: <%# custID %>
```

- С колекция:

```
Orders: <asp:ListBox id="List1" datasource='<%# myArray %>'
runat="server">
```

- С израз:

```
Contact: <%# ( customer.First Name + " " + customer.LastName )
%>
```

- С резултат от метод:

```
Outstanding Balance: <%# GetBalance(custID) %>
```

Как работи методът DataBind(...)?

Въпреки че изглежда подобен на **<% Response.Write(customer.Name) %>** *или* **<%= customer.Name %>**, поведението на метода е различно. Докато първите два блока се изпълняват когато страницата генерира HTML от **Render(…)** метода, ASP.NET изразите за свързване с данни се изпълняват при извикването на **DataBind(…)**. Ако методът не се извика, целият регион **<%#... %>** се игнорира.

DataBind(…) е метод на класа **Page** и на сървър контролите. Когато се извика **DataBind(…)** на родителската контрола, той се извиква каскадно и за всички нейни деца. Извикването на **DataBind(…)** на вградения обект **Page** (**Page.DataBind(…)** или по-просто **DataBind(…)**), предизвиква оценяването на всички изрази (**<%#... %>**) за свързване с данни. Най-често **DataBind(…)** се извиква от **Page_Load** събитието:

```
void Page_Load(Object sender, EventArgs e)
{
    Page.DataBind();
}
```

Може да се използва почти навсякъде в декларативната част на **.aspx** страница, стига да се връща подходящ тип данни. В някои случаи се налага преобразуване на данните.

Свързване на контроли с данни – пример

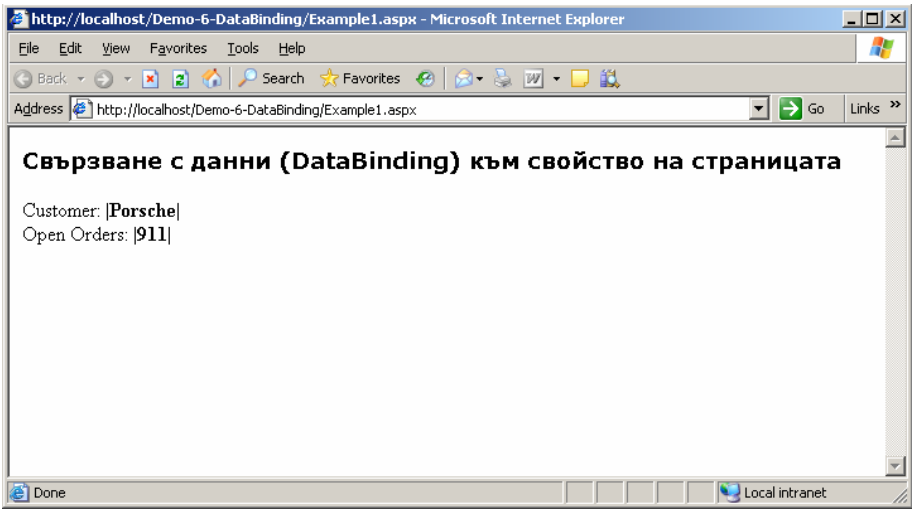
Първият пример за свързване с данни, който ще разгледаме, работи директно със свойства на страницата. Ето кода:

```
<html>
<body>
    <h3><font face="Verdana">Свързване с данни (DataBinding)
      към свойство на страницата</font></h3>
    <form runat=server ID="FormExample1">
        Customer: <b>|<%# custID %>|</b><br>
        Open Orders: <b>|<%# orderCount %>|</b>
    </form>
</body>
</html>
```

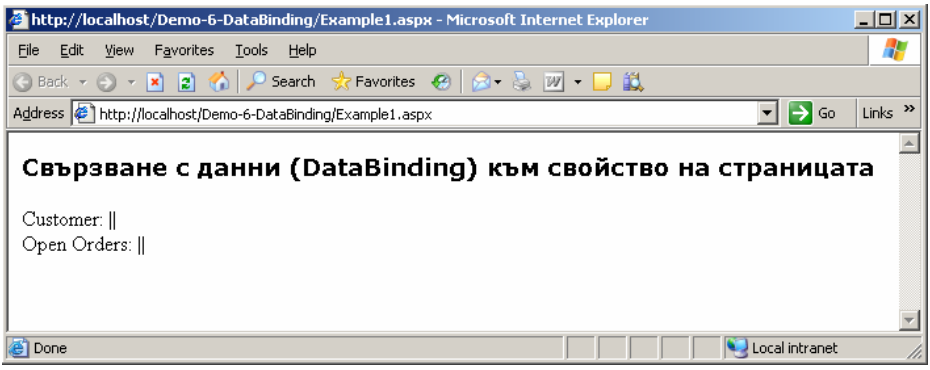
Ето и кода в code-behind страницата:

```
void Page_Load(Object sender, EventArgs e)
{
   Page.DataBind();
}

string custID
{
   get
   {
      return "Porsche";
   }
}

int orderCount
{
   get
   {
      return 911;
   }
}
```

Ето го резултатът:



Важно е да извикаме метода **`Page.DataBind()`**. Ако го закоментираме, ето какво се случва:



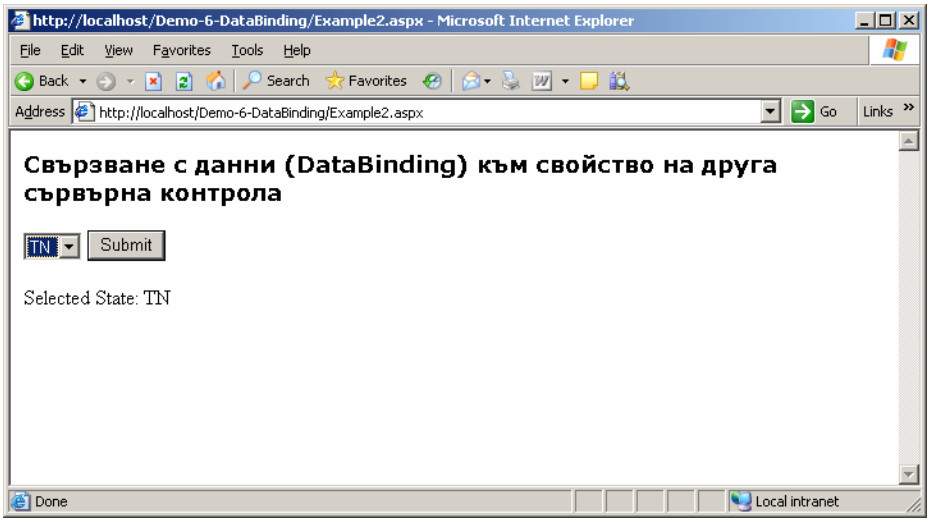
Както виждаме, нищо не е изписано между вертикалните черти.

В следващия пример една контрола ще достъпва данни от друга контрола. В конкретния случай етикет (**`Label`**) ще изписва избрания щат от падащ списък. Ето го кода:

```
<html>
  <body>
    Свързване с данни (DataBinding) към свойство на друга
       сървърна контрола
    <form runat="server" ID="FormExample2">
      <asp:DropDownList id="DropDownListState" runat="server">
        <asp:ListItem>CA</asp:ListItem>
        <asp:ListItem>IN</asp:ListItem>
        <asp:ListItem>KS</asp:ListItem>
        <asp:ListItem>MD</asp:ListItem>
        <asp:ListItem>MI</asp:ListItem>
        <asp:ListItem>OR</asp:ListItem>
```

```
            <asp:ListItem>TN</asp:ListItem>
            <asp:ListItem>UT</asp:ListItem>
        </asp:DropDownList>
        <asp:Button Text="Submit" OnClick="ButtonSubmit_Click"
          runat="server" ID="ButtonSubmit" Name="Button1" />
        Selected State:
        <asp:Label text='<%# StateList.SelectedItem.Text %>'
          runat=server ID="LabelState" Name="LabelState"/>
      </form>
    </body>
</html>
```

Разбира се, в **`Page_Load`** метода извикваме **`DataBind(…)`**. Резултатът е:

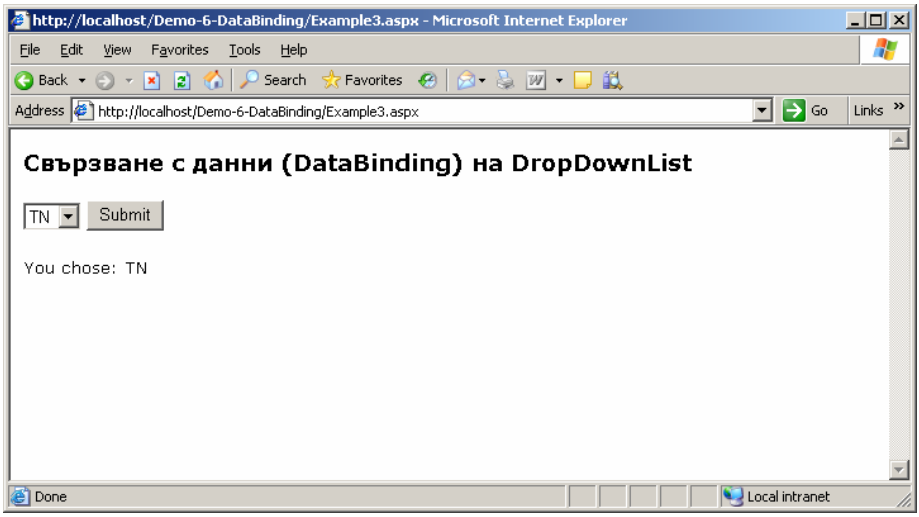


В третия пример ще заредим падащ списък от масив. Нека имаме уеб форма с падащ списък на нея. За краткост ще дадем кода само от code-behind класа:

```
void Page_Load(Object Sender, EventArgs E)
{
  if (!Page.IsPostBack)
  {
    ArrayList values = new ArrayList();

    values.Add ("IN");
    values.Add ("KS");
    values.Add ("MD");
    values.Add ("MI");
    values.Add ("OR");
    values.Add ("TN");

    DropDownListCountries.DataSource = values;
    DropDownListCountries.DataBind();
```

```
   }
}

void ButtonSubmit_Click(Object sender, EventArgs e)
{
   LabelChosen.Text = "You chose: " +
      DropDownListCountries.SelectedItem.Text;
}
```

От този код падащият списък се зарежда с няколко щата. Вторият метод изписва в етикет избрания щат.

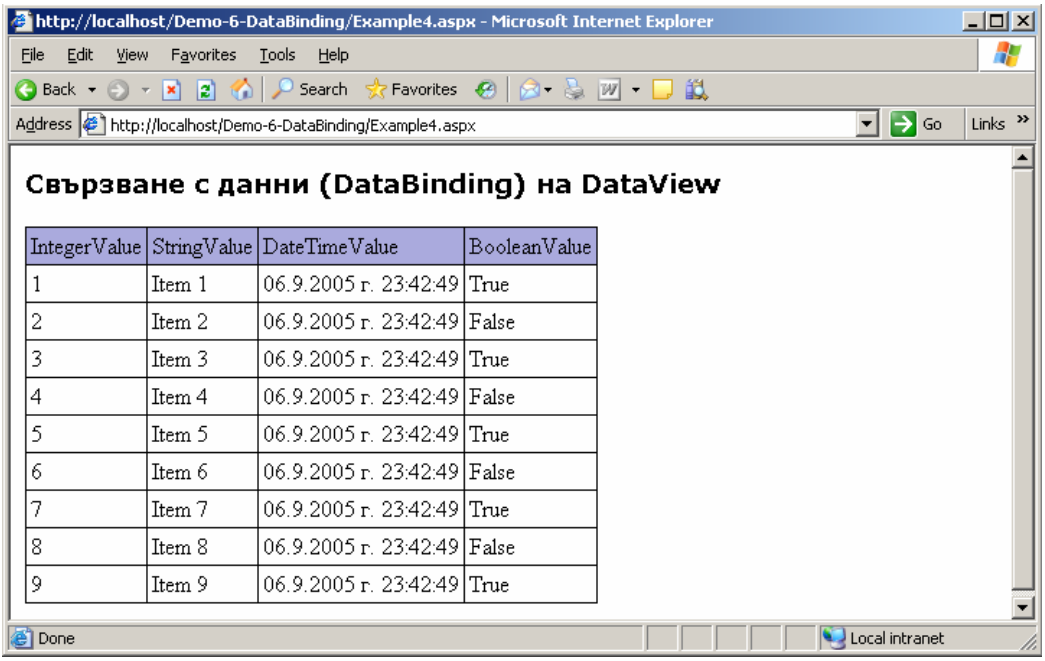


Сега ще демонстрираме показване на данни в **DataGrid** от табличен източник на данни (в случая **DataView**):

```
<%@ Import namespace="System.Data" %>

<html>
<head>
   <script language="C#" runat="server">
      void Page_Load(Object sender, EventArgs e )
      {
         if (!Page.IsPostBack)
         {
            DataTable dt = new DataTable();
            DataRow dr;
            dt.Columns.Add(new DataColumn("IntegerValue",
               typeof(Int32)));
            dt.Columns.Add(new DataColumn("StringValue",
               typeof(string)));
            dt.Columns.Add(new DataColumn("DateTimeValue",
               typeof(DateTime)));
            dt.Columns.Add(new DataColumn("BooleanValue",
```

```
            typeof(bool)));

         for (int i= 1; i <= 9; i++)
         {
            dr = dt.NewRow();
            dr[0] = i;
            dr[1] = "Item " + i.ToString();
            dr[2] = DateTime.Now;
            dr[3] = (i % 2 != 0) ? true : false;

            dt.Rows.Add(dr);
         }

         dataGridExample.DataSource = new DataView(dt);
         dataGridExample.DataBind();
      }
   }
   </script>
</head>
<body>
   <h3><font face="Verdana">Свързване с данни (DataBinding) на
      DataView</font></h3>
   <form runat=server ID="FormExample4">
      <asp:DataGrid id="dataGridExample" runat="server" … />
   </form>
</body>
</html>
```

При компилирането му се показват данните в браузъра:



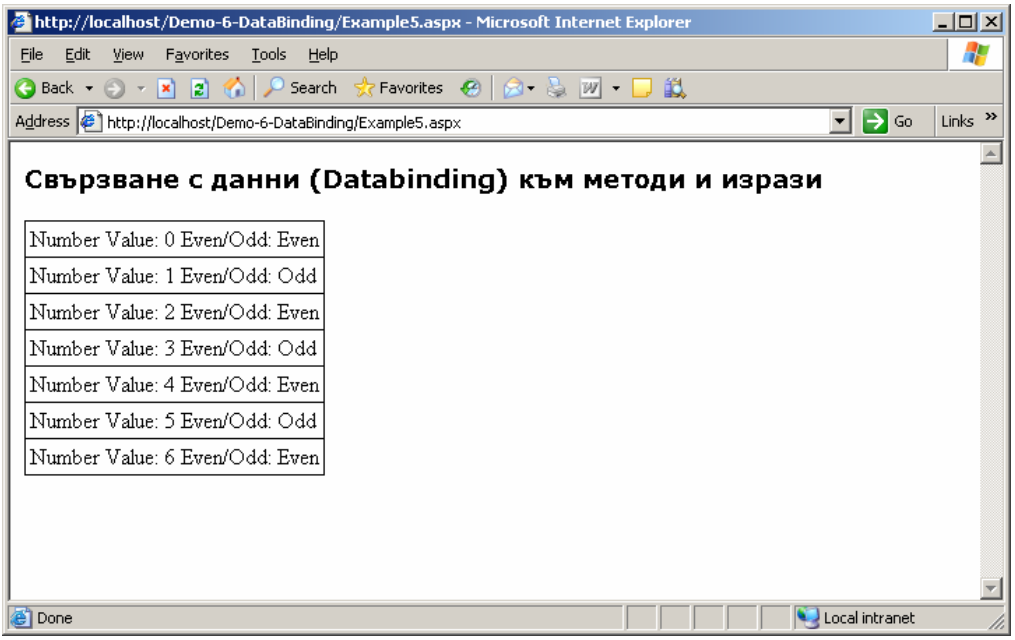
IntegerValue	StringValue	DateTimeValue	BooleanValue
1	Item 1	06.9.2005 г. 23:42:49	True
2	Item 2	06.9.2005 г. 23:42:49	False
3	Item 3	06.9.2005 г. 23:42:49	True
4	Item 4	06.9.2005 г. 23:42:49	False
5	Item 5	06.9.2005 г. 23:42:49	True
6	Item 6	06.9.2005 г. 23:42:49	False
7	Item 7	06.9.2005 г. 23:42:49	True
8	Item 8	06.9.2005 г. 23:42:49	False
9	Item 9	06.9.2005 г. 23:42:49	True

В последния пример ще направим свързване на данни с изрази (expressions) и методи, които ще извикваме параметрично:

```
<html>
<head>
  <script language="C#" runat="server">
    void Page_Load(Object Src, EventArgs E)
    {
      if (!Page.IsPostBack)
      {
        ArrayList values = new ArrayList();
        values.Add (0);
        values.Add (1);
        values.Add (2);
        values.Add (3);
        values.Add (4);
        values.Add (5);
        values.Add (6);

        DataListExample.DataSource = values;
        DataListExample.DataBind();
      }
    }

    String EvenOrOdd(int number)
    {
      if ((number % 2) == 0)
        return "Even";
      else
        return "Odd";
    }
   </script>
</head>
<body>

  Свързване с данни (Databinding) към методи и изрази

  <form runat=server ID="FormExample5">
    <asp:DataList id="DataListExample" runat="server" … >
       <ItemTemplate>
         Number Value: <%# Container.DataItem %>
         Even/Odd: <%# EvenOrOdd((int) Container.DataItem) %>
       </ItemTemplate>
     </asp:datalist>
   </form>
</body>
</html>
```

Този път данните се показват в **DataList**, в шаблона, на който се показват данните от целочислен масив и извикванията от метод. На метода се подава поредната целочислена стойност. Забележете гъвкавостта на

работата с данни – можем да вземем поредния запис от база от данни и на негова основа да покажем допълнителни стойности или изцяло променени данни.



Работа с бази от данни от ASP.NET

В практиката при почти всички уеб приложения се налага работа с данни. Типичният сценарий включва визуализация на таблични данни, идващи от таблици в базата данни, както и добавяне на нови записи и редактиране и изтриване на съществуващи. Такива приложения обикновено се изграждат чрез свързване на ASP.NET уеб форми с ADO.NET.

Преди да изясним по какъв начин можем да изграждаме уеб приложения, използващи релационни бази от данни, нека си припомним основните концепции от ADO.NET. Ще направим само общ обзор на взаимодействието между ASP.NET и ADO.NET. За повече подробности за ADO.NET и работата с бази от данни, разгледайте главата, посветена специално на тази тема.

Обзор на ADO.NET

При създаването на уеб сайтове, които трябва да поддържат хиляди едновременни посещения от хиляди потребители, ще са нужни същия брой отворени връзки към базата от данни. Дори сървърите, отговарящи за поддръжката на базата от данни, да успеят да издържат на това натоварване, скоростта, с която ще работи приложението, ще е нетърпимо бавна. Затова е силно препоръчително при работа с бази от данни с ASP.NET да се използва несвързаният модел.

Обекти за работа с бази от данни

В главата за ADO.NET тези обекти са подробно обяснени, затова тук само ще ги споменем и ще отбележим как се използват в ASP.NET.

- **`Connection`** – връзка към базата от данни.
- **`Command`** – команда, служеща за изпълняване на заявки върху базата от данни и за извличане на данни.
- **`DataReader`** – четец на данни, върнати като резултат от заявка от базата от данни.
- **`DataSet`** – кеш на част от базата от данни в паметта. Съдържа таблици, релации, ограничения и т.н.
- **`DataAdapter`** – средство за извличане на данните от базата и обновяването им чрез **`DataSet`** обекти.

Визуализиране на данни

Почти всяко уеб приложение, което ползва база от данни, има нужда да представи тези данни на потребителя. Когато се отнася до единични полета (пр. потребителско име или дата) се използват етикети или литерали. Какво обаче да правим, ако искаме да покажем списъка на всички потребители с техните детайли под формата на таблица? Или ако искаме да ги покажем в падащо меню? За да реализираме тези и много други манипулации можем да използваме т.нар. свързани контроли (bound controls). Те играят важна роля в разработката на уеб приложения, защото позволяват бърза и интуитивна работа.

Ще разгледаме два вида свързване на данните – просто и сложно, както и най-важните свързани контроли.

Свързване на данни (data binding)

Свързването на данни е процесът на динамично извличане на данни от зададен източник и визуализирането им чрез подходящи контроли.

За различните контроли източникът на данни се задава чрез различни свойства - **`Text`**, **`DataSource`**, **`DataTextField`**. След малко ще обясним как се използват тези свойства при различните свързани контроли.

Трябва да отбележим, че в някои случаи (например при използването на свойството **`DataSource`**), свързване не се извършва, преди да се извика методът **`DataBind()`**.

Източниците на данни за свързаните контроли могат да са разнородни, не само данни от бази от данни. Източник на данни може да е всеки обект от клас, имплементиращ интерфейса **`ICollection`**. Имплементацията на този интерфейс дава всичко необходимо, за да се извърши свързването на данните. Като резултат от това, можем да използваме като източници на данни за свързани контроли всяка една от следните структури:

- Класове от .NET Framework, които имплементират **ICollection**: масиви, списъци (сортирани или свързани), хеш таблици, стекове, опашки, речникови колекции.
- Потребителски класове, имплементиращи интерфейса **ICollection** или някой производен на него (примерно **IList**).
- Класове, свързани с работата с бази данни – **DataTable** и **DataSet**. Тъй като обект от тип **DataSet** може да съдържа много **DataTable** обекти след като укажем на **DataSource** свойството **DataSet** обекта, трябва да зададем на свойството **DataMember** името на таблицата, която искаме да свържем.
- Филтрирани подмножества от редовете на една **DataTable** таблицата: обекти от тип **DataView**.

Не можем директно да зададем като източник на данни за свързване XML документ. Трябва да заредим съдържанието на документа в една от споменатите по-горе структури, за да се възползваме от свързването на данни.

Просто свързване

Простото свързване указва връзка между някакви данни и свойство на някоя контрола. Тази връзка се задейства при извикването на метода **DataBind()** на форма или контрола, когато свързващият израз се оценява и се прилага.

Свързващ израз представлява всеки текст, заграден в таговете **<%#** и **%>**. Свързващи изрази можем да поставяме навсякъде в **.aspx** (**.ascx**) файла на уеб форма/контрола. Най-често те заместват стойността на някой атрибут на контрола. Ограждат се с единични кавички, за да се отличават от атрибутите, които са с двойни кавички:

```
<asp:Button ID="btnName" Runat="server"
  Text='<%# "Бай Иван" %>' />
```

Ако създадем нова уеб форма, поставим в нея гореописания бутон и натиснем **[F5]**, ще забележим, че като текст на бутона не се показва нищо. Това е така, защото не сме извикали метода **DataBind()**. За да проработи горният пример, трябва да извикаме този метод, примерно при обработчика на събитието **Load** на формата:

```
private void Page_Load(object sender, System.EventArgs e)
{
   this.DataBind();
}
```

Горният пример не би се използвал в практиката, защото бихме постигнали същия ефект директно с **Text="Бай Иван"**. Ползите от такъв синтак-

сис се виждат в случаите, когато за свързващ израз указваме по-сложен израз. В израза могат да се викат методи на езика, на който се компилира страницата, например:

```
<asp:textbox id="txtFirstName" runat="server"
  Text='<%# GetData("FirstName") %>' />
<asp:textbox id="txtLastName" runat="server"
  Text='<%# GetData("LastName") %>' />
```

Трябва да направим уточнение, че методите, участващи в израза, трябва да са достъпни във формата. Формата е наследник на code-behind класа и следователно не можем да извикаме от нея частен (**private**) метод на code-behind класа. Затова щом дефинираме методи, които искаме да извикваме от формата, трябва да им укажем видимост **public**, или **protected**. Ето пример:

```
public string GetData( string fieldName )
{
   switch (fieldName)
   {
      case "FirstName":
      {
         return "Иван";
      }
         break;
      case "LastName":
      {
         return "Иванов";
      }
         break;
      default:
      {
         return "Unknown";
      }
         break;
   }
}
```

Сложно свързване

Много често ни се налага да визуализираме голямо количество данни, извлечени от база от данни. Сложното свързване представлява свързване на множество редове/свойства с една контрола. Използва се предимно в списъчните и итериращите контроли, които ще разгледаме по-долу.

Контроли за показване на данни

Можем условно да разделим контролите за показване на данни в две групи - списъчни и итериращи.

Списъчни контроли

Списъчните контроли са **DropDownList**, **CheckBoxList**, **RadioButtonList** и **ListBox**. Фигурирането на думата List (списък) в името им, показва че служат за представяне на данните под формата на списък. Нека разгледаме какви са общите неща между тях.

Базовият клас ListControl

Класовете на списъчните контроли произлизат от един и същ абстрактен базов клас – **ListControl**. Той осигурява голяма част от функционалността на списъчните контроли.

Всяка списъчна контрола съдържа колекция **Items**. Тази колекция отговаря за елементите на списъка. Всеки елемент на списъка е от тип **ListItem** и има три свойства, които го характеризират – **Text**, **Value** и **Selected**. Полето **Text** съдържа текста, който да се покаже. Полето **Value** съдържа стойността на съответния елемент. Примерно можем да покажем списък с имената на държавите България, Германия, Русия, САЩ, и да използваме за ключова стойност техните държавни кодове – BGR, DEU, RUS, USA. Полето **Selected** съдържа булева стойност, показваща дали съответният елемент на списъка е избран.

Основната функционалност, която предоставят списъчните контроли, е възможност за избор от елементите на списъка.

Текущ избран елемент в списъчни контроли

За работа с избраните елементи се използват свойствата **SelectedIndex**, **SelectedItem** и **SelectedValue**:

- **SelectedIndex** връща индекса на първия избран елемент от списъка. Ако няма такъв, връща **-1**. Стойността на това поле може да се задава програмно.
- **SelectedItem** връща първия избран елемент от списъка. Ако няма такъв, връща **null**.
- **SelectedValue** връща стойността на първия избран елемент от списъка. Ако няма такъв, връща **null**. Може да използваме това поле, когато искаме да зададем програмно на някоя списъчна контрола избран елемент, но не знаем на коя позиция се намира в списъка. Например, ако имаме списък с всички държави и искаме България да е избрана, но не знаем на коя позиция се намира. Тогава задаваме като стойност на полето **SelectedValue** – "BGR".

Всички списъчни контроли имат събитие **SelectedIndexChanged**. То се предизвиква при промяна на индекса на първия избран елемент.

Елементите на списъчните контроли могат да се дефинират основно по три начина: декларативно (директно в **.aspx**/**.ascx** файла), динамично (като се добавят един по един в изпълним код), и чрез свързване (като съответ-

ната списъчна контрола се свърже с някой източник на данни). Ще демонстрираме всеки един от трите метода.

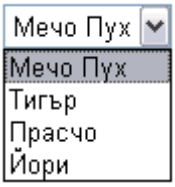
Списъчни контроли и свързване с данните

Всички списъчни контроли имат едни и същи свойства, отговарящи за свързването с източници на данни - **DataSource**, **DataTextField**, **DataValueField**, **DataTextFormatString** и **DataMember**:

- **DataSource** определя източника на данни.
- **DataTextField** определя стойностите на кое поле от източника на данни да се използват за стойности на полето **Text** за елементите на списъка.
- **DataValueField** определя стойностите на кое поле от източника на данни да се използват за стойности на полето **Value** за елементите на списъка.
- **DataTextFormatString** определя какъв форматиращ низ да се използва за визуализиране на текста. В случай, че източникът на данни е от тип **DataSet**, съдържащ повече от един **DataTable** обект, се използва полето **DataMember**, което задава коя точно таблица да се използва.

Контролата DropDownList

Контролата **DropDownList** показва данните под формата на падащ списък, от който може да се избира само един елемент:



Всеки един от елементите на списъка е обект от тип **ListItem**. Нека покажем как става тяхното дефиниране за **DropDownList** контролата в горния пример. Има три основни начина: декларативно (статично), динамично и чрез свързване на данни (data binding).

Декларативно (статично) задаване на елементите в списъчни контроли

Декларативното (известно още като статично) задаване на елементите в списъчни контроли е най-простият вариант да заредим данни в списъчна контрола. То става чрез дефинираме елементите директно в **.aspx** (**.ascx**) файла:

```
<asp:DropDownList ID="ddlList" Runat="server">
```

```
   <asp:ListItem Value="1">Мечо Пух</asp:ListItem>
   <asp:ListItem Value="2">Тигър</asp:ListItem>
   <asp:ListItem Value="3">Прасчо</asp:ListItem>
   <asp:ListItem Value="4">Йори</asp:ListItem>
</asp:DropDownList>
```

Динамично задаване на елементите в списъчни контроли

При динамичното задаване на елементите в списъчни контроли се използва свойството **Items**.

Ето един пример. Дефинираме контролата в **.aspx** (**.ascx**) файла:

```
<asp:DropDownList ID="ddlList" Runat="server">
</asp:DropDownList>
```

След това динамично добавяме елементите в кода:

```
ddlList.Items.Add(new ListItem("Мечо Пух", "1"));
ddlList.Items.Add(new ListItem("Тигър", "2"));
ddlList.Items.Add(new ListItem("Прасчо", "3"));
ddlList.Items.Add(new ListItem("Йори", "4"));
```

Задаване на елементите в списъчни контроли чрез свързване на данни

Свързването на данни със списъчна контрола се използва най-често, когато данните идват от базата данни. То се реализира малко по-сложно в сравнение със статичното и динамичното задаване на елементите.

Започваме с указване на източника на данни, като ще използваме два различни типа. Първият ще бъде масив от потребителски обекти, а вторият **DataSet**, съдържащ **DataTable**.

Потребителският клас, от който ще се състои масивът, изглежда така:

```
public class Character
{
   private string name;
   private long id;
   public string Name
   {
      get { return name; }
      set { name = value; }
   }
   public long ID
   {
      get { return id; }
      set { id = value; }
   }
```

```
   public Character(string name, long id)
   {
      this.name = name;
      this.id = id;
   }
}
```

Дефинираме контролата в **.aspx** или **.ascx** файл:

```
<asp:DropDownList ID="ddlList" Runat="server"
   DataTextField="Name" DataValueField="ID">
</asp:DropDownList>
```

Изкуствено ще създадем масив от елементи **Character** (на практика този масив може е извлечен от база от данни):

```
Character[] bookCharacters = new Character[]
{
   new Character("Мечо Пух", 1),
   new Character("Тигър", 2),
   new Character("Прасчо", 3),
   new Character("Йори", 4)
};
ddlList.DataSource = bookCharacters;
ddlList.DataBind();
```

В последните два реда от примера, масивът **bookCharacters** се задава като източник на данни за **DropDownList** контролата и се извиква методът **DataBind()**, за да се свържат данните с нея.

За да демонстрираме свързване с **DataSet** обект, ще се наложи и него да създадем изкуствено. Нека дефинираме отделен метод, който връща обект от тип **DataSet**:

```
public DataSet GetDataSource()
{
   DataSet dataSource = new DataSet();
   DataTable charactersTable = new DataTable("Characters");

   Characters.Columns.Add("ID", typeof(long));
   Characters.Columns.Add("Name", typeof(string));

   DataRow row1 = charactersTable.NewRow();
   DataRow row2 = charactersTable.NewRow();
   DataRow row3 = charactersTable.NewRow();
   DataRow row4 = charactersTable.NewRow();

   row1["Name"] = "Мечо Пух";
   row1["ID"] = 1;
```

```
    row2["Name"] = "Тигър";
    row2["ID"] = 2;
    row3["Name"] = "Прасчо";
    row3["ID"] = 3;
    row4["Name"] = "Йори";
    row4["ID"] = 4;

    charactersTable.Rows.Add(row1);
    charactersTable.Rows.Add(row2);
    charactersTable.Rows.Add(row3);
    charactersTable.Rows.Add(row4);

    dataSource.Tables.Add(charactersTable);

    return dataSource;
}
```

Дефинираме контролата в **.aspx** или **.ascx** файла:

```
<asp:DropDownList ID="ddlList" Runat="server"
    DataSource='<%# GetDataSource() %>' DataTextField="Name"
    DataValueField="ID" DataMember="Characters">
</asp:DropDownList>
```

В примера по-горе като стойност на свойството **DataSource** сме задали свързващ израз, който извиква функцията **GetDataSource()**. Свързващ израз може да се задава само за това поле на списъчна контрола. Указали сме също и стойността на свойството **DataMember** да е "Characters" – името на таблицата от **DataSet** обекта, която служи за източник на данни. В случая **DataSet** обектът има само една таблица и полето **DataMember** може да бъде пропуснато, но сме го дали за пълнота.

Контролата CheckBoxList

Тази контрола показва данните под формата на списък от **CheckBox** контроли, от който могат да се избират произволен брой елементи. Ето как изглежда тя:

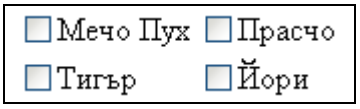
Начините, по които могат да се зададат елементите на списъка, са аналогични на тези за **DropDownList** контролата. Допълнителните характеристики за **CheckBoxList** се определят чрез полетата **RepeatColumns**, **RepeatDirection** и **RepeatLayout**:

- Чрез свойството **RepeatColumns** се задава в колко колони да се покаже списъкът (по подразбиране в една).
- Свойството **RepeatDirection** определя в каква посока да се подреждат елементите на списъка - **Vertical** (по подразбиране) или **Horizontal**.

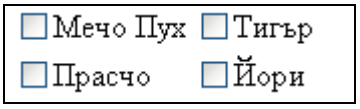
Нека дефинираме контролата така:

```
<asp:CheckBoxList ID="chkCharactersList" Runat="server"
     RepeatColumns="2" RepeatDirection="Vertical">
   <asp:ListItem Value="1">Мечо Пух</asp:ListItem>
   <asp:ListItem Value="2">Тигър</asp:ListItem>
   <asp:ListItem Value="3">Прасчо</asp:ListItem>
   <asp:ListItem Value="4">Йори</asp:ListItem>
</asp:CheckBoxList>
```

В този случай резултатът ще бъде следният:



Ако свойството **RepeatDirection** има стойност **Horizontal**, вместо **Vertical**, то резултатът ще е следният:

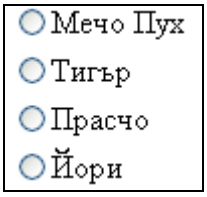


Свойството **RepeatLayout** отговаря за начина, по който се представят елементите на списъка. То може да приема само две стойности – **Flow** и **Table**. Стойността му по подразбиране е **Table**. Ако стойността е **Flow[,]**, за да се представят елементите на различни редове (един под друг), след последния елемент на всеки ред се поставя **
, за да се премине на следващия. Ако стойността е **Table, елементите се представят в таблична структура.

Контролата RadioButtonList

Разликите между **CheckBoxList** и **RadioButtonList** са в начина на представяне на данните и в броя на елементите, които могат да се избират едновременно.

При **RadioButtonList** елементите на списъка се представят така:

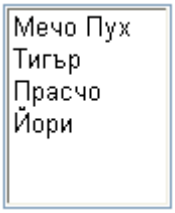


Само един елемент от списъка може да бъде избран.

Начините за дефиниране на елементите на списъка и свойствата на контролата са същите, както при **CheckBoxList**.

Контролата ListBox

Тази контрола показва данните под формата на списък, поставен в кутия.



Има две свойства, които характеризират тази контрола: **Rows** и **SelectionMode**:

- Стойността на полето **Rows** определя от колко реда се състои кутията. Ако елементите на списъка са повече от тази стойност, отдясно на кутията се появява плъзгаща се лента (**ScrollBar**).
- Свойството **SelectionMode** може да приема само две стойности – **Single** и **Multiple**. Ако стойността му е **Single**, то от списъка може да се избира само един елемент. В противен случай може да се извършва множествена селекция.

Итериращи контроли

Списъчните контроли предоставят базовата функционалност, необходима на едно уеб приложението да комуникира успешно с потребителя. Чрез тях данните се показват на потребителя и той може да направи избор от тях.

В много случаи е удачно тези данни да се представят в табличен вид. Например, ако работим с база от данни и искаме нашето приложение да визуализира всички данни от дадена таблица.

Старият начин за итерация

Нуждата от визуализиране на данни в таблична форма не е нещо ново. В класическото ASP (преди появата на ASP.NET) се ползваше следния начин на реализация:

```
<table border="1" cellpadding="0" cellspacing="0">
  <tr>
    <%
      int dataItemsCount = Data.Tables[0].Rows.Count;
      int dataColumnsCount = Data.Tables[0].Columns.Count;
```

```
        for(int i=0; i < dataColumnsCount; i++)
        {
     %>
     <td align="center"
            style="background-color: #00AAFF; padding: 5px;">
        <b>
           <%= Data.Tables[0].Columns[i].ColumnName %>
        </b>
     </td>
     <% }%>
   </tr>
     <%
        for(int i = 0; i< dataItemsCount; i++)
        {
     %>
   <tr style="background-color: white;">
     <%
          for(int j = 0; j< dataColumnsCount; j++)
          {
     %>
     <td style="padding: 5px;">
        <%= Data.Tables[0].Rows[i][j].ToString()%>
     </td>
     <% }%>
   <tr>
     <% }%>
</table>
```

В примера по-горе сме използвали обекта **Data**, който е от тип **DataSet**. Тъй като и за напред ще ни се наложи да го използваме в различни примери, нека зададем следната дефиниция за **Data**:

```
private DataSet dsData;

public DataSet Data
{
   get
   {
      return dsData;
   }
}

private void Page_Load(object sender, System.EventArgs e)
{
   GenerateDataSet();
}

private void GenerateDataSet()
{
```

```
    dsData = new DataSet();
    DataTable dtData = new DataTable("Characters");

    dtData.Columns.Add("Character First Name");
    dtData.Columns.Add("Character Last Name");
    dtData.Columns.Add("Character Birth Date", typeof(DateTime));
    dtData.Columns.Add("Character Age", typeof(Int32));

    DataRow drData = dtData.NewRow();

    drData["Character First Name"] = "Мечо";
    drData["Character Last Name"] = "Пух";
    drData["Character Birth Date"] = new DateTime(1971,4,1);
    drData["Character Age"] = 35;
    dtData.Rows.Add(drData);

    drData = dtData.NewRow();

    drData["Character First Name"] = "Прасчо";
    drData["Character Last Name"] = "Свински";
    drData["Character Birth Date"] = new DateTime(1978,5,11);
    drData["Character Age"] = 28;
    dtData.Rows.Add(drData);

    drData = dtData.NewRow();

    drData["Character First Name"] = "Тигър";
    drData["Character Last Name"] = "Бенгалски";
    drData["Character Birth Date"] = new DateTime(1984,8,12);
    drData["Character Age"] = 21;
    dtData.Rows.Add(drData);

    drData = dtData.NewRow();

    drData["Character First Name"] = "Йори";
    drData["Character Last Name"] = "Магарисченко";
    drData["Character Birth Date"] = new DateTime(1955,4,30);
    drData["Character Age"] = 51;
    dtData.Rows.Add(drData);

    dsData.Tables.Add(dtData);
}
```

Дефинираме обекта **Data** като свойство на страницата, което връща член-променливата **dsData**. По време на зареждане на страницата (**Page_Load**) извикваме метода **GenerateDataSet()**. Той инициализира **dsData** с примерни данни. За простота тук данните не се вземат от база от данни, но това е без значение за целите на демонстрацията.

Как работи примерът?

Да се върнем обратно към примера. Ако сте разработвали приложения с ASP, този пример сигурно ви се струва донякъде познат. За тези, които тепърва започват да се запознават с разработка на уеб приложения с ASP.NET, кодът сигурно изглежда изключително объркващ. Няма да се задълбочаваме в детайли, а ще обясним само ключовите части на примера.

Както сте забелязали, тук по особен начин се смесват сървърни тагове (**<% %>**) и обикновен HTML. Нека разгледаме следния отрязък код:

```
<b>Character List</b> <br />
<% for(int i = 0 ; i < 10; i++)
   {
%>
   <b>  <% if((i % 2)> 0) { %> <i> <% }%>
     Character <%= (i+1)%>
    <% if((i % 2)> 0) { %> </i> <% }%> </b>
     <br />
<% } %>
<b>Total : 10</b>
```

В резултат на интерпретирането на този код, в уеб браузъра на клиента ще пристигне следният HTML:

```
<b>Character List</b> <br />
<b>
   Character 1
</b>
   <br />
<b>   <i>
   Character 2
</i> </b>
   <br />
<b>
   Character 3
</b>
   <br />
<b>   <i>
   Character 4
</i> </b>
   <br />
<b>
   Character 5
</b>
   <br />
<b>   <i>
   Character 6
</i> </b>
   <br />
<b>
```

```
   Character 7
</b>
   <br />
<b>   <i>
   Character 8
</i> </b>
   <br />
<b>
   Character 9
</b>
   <br />
<b>   <i>
   Character 10
</i> </b>
<b>Total : 10</b>
```

В крайна сметка браузърът ще покаже следния списък:

Hero List
Hero 1
Hero 2
Hero 3
Hero 4
Hero 5
Hero 6
Hero 7
Hero 8
Hero 9
Hero 10
Total : 10

Примерният код се интерпретира така: за всяка стойност на `i` от 0 до 9 се изпълнява тялото на цикъла. В него всичко, което не е заградено в сървърни тагове (`<% %>`) остава непроменено, а всичко заградено в сървърни тагове се интерпретира.

Този опростен пример демонстрира как може да се реализира повторение на HTML елементи. В първоначалния пример вместо фиксирана стойност 10 използваме **`dataColumnsCount`** и **`dataItemsCount`**.

Този начин на реализация на представяне на данни в табличен вид е твърде непрактичен, но преди ASP.NET не е имало друга алтернатива. Основните му недостатъци идват от необходимостта от смесването на процедурен код и HTML, което води до нечетливост на написаното и затруднения в поддръжката.

Нуждата от начин за представяне на данните в табличен вид е довела до създаването на контролите **`DataGrid`**, **`DataList`** и **`Repeater`**.

Сходства между итериращите контроли

Всички итериращи контроли съдържат списък с елементи, отговорни за генерирането на изходния HTML.

DataGrid показва записите в HTML таблица (чрез тага **<table>**), където всеки елемент се представя в отделен ред. Класът **DataGridItem** е предназначен да визуализира табличен запис и затова е наследник на класа **TableRow**.

Аналогично **DataList** е съставен от елементи от тип **DataListItems**. Класът **Repeater**, от друга страна, позволява пълна настройка на изходния HTML. Затова класът за елементите му **RepeaterItem** не е наследник на **TableRow**.

При извикване на метода **DataBind()** се преминава през всички записи на свойството **DataSource**. За всеки запис се създава нова инстанция от тип **DataWebControlNameItem** и записът се свързва със свойството й **DataItem**.

Събития на итериращите контроли за свързване с данните

Итериращите контроли поддържат няколко общи събития, касаещи процеса на свързването на данните:

- Събитието **ItemCreated** се активира за всеки нов **DataWebControlNameItem** добавен към контролата, преди още да е инициализирано свойството **DataItem**. Събитието **ItemDataBound** се случва веднага след инициализацията на свойството **DataItem**. А **ItemCommand** събитието се активира при всяка команда от **Button** or **LinkButton** в итериращата контрола.
- Друга важна обща характеристика на итериращи контроли е, че всички позволяват използването на шаблони. Контролите **DataList** и **Repeater** задължително изискват шаблони, докато при **DataGrid** използването им е незадължително.
- **DataGrid** и **DataList** са наследници на класа **WebControl**, докато **Repeater** е наследник на класа **Control**. Класът **WebControl** има множество свойства свързани с визуализацията: **BackColor**, **ForeColor**, **CssClass**, **BorderStyle** и др. **Repeater** контрола не поддържа директно тези свойства, но аналогични форматиращи настройки могат да бъдат указвани чрез шаблоните му.

Контролата DataGrid

От гледна точка на вградени възможности, **DataGrid** е най-мощната от итериращите контроли. За сметка на това, тя не е гъвкава по отношение на генерирания HTML код. Той винаги генерира HTML таблици, като за всяка свързана колона се създава ред чрез тага **<tr>** и за всяко поле от записа, се създава колона чрез тага **<td>**.

Сред вградените възможности на **DataGrid** контролата са сортиране, страниране и редакция на данните директно в таблицата. Примерно чрез указване на свойството **AllowSorting = true** и малко допълнителен код, лесно може да се предостави на потребителя средство за сортиране.

С **DataGrid** можем много бързо да реализираме показване на данни в ASP.NET уеб страница. Само трябва да поставим контрола в страницата, да укажем **DataSource** и да извикваме **DataBind()**. **DataGrid** има свойство **AutoGenerateColumns**, с което може да укажем дали колоните се генерират автоматично, или дали само ще задаваме кои от тях да се покажат и по какъв начин.

Всяка колона в **DataGrid** е инстанция на клас, наследник на **DataGridColumn**. Вградените типове колони са:

- **BoundColumn** – колона, свързана с поле от източника на данни. Показва данните под формата на обикновен текст.
- **ButtonColumn** – колона, показваща бутон.
- **EditColumn** – колона за редакция на данни.
- **HyperLinkColumn** – колона, показваща хипервръзка, като текста и URL-то могат да бъдат от отделни полета на източника на данни.
- **TemplateColumn** – колона шаблон, чрез която може да се генерира произволен изходен HTML. Има шаблони за различните части на таблицата: **ItemTemplate**, **HeaderTemplate**, **FooterTemplate** и **EditItemTemplate**.

Производителността на **DataGrid** понякога може да е проблем, тъй като при голям обем данни размерът ViewState на контролата става значителен. Ако ViewState бъде изключен, то това ще е за сметка на възможностите за сортиране, страниране и редактиране.

Контролата DataList

Необходимостта от **DataList** възниква, когато представянето на данни в HTML таблица с по един запис на ред е неудачно. Понякога може да искаме да покажем повече от един запис на ред или да решим да използваме **** вместо **<table>** тагове.

При **DataList** концепцията за "колони" не присъства. Всички настройки се задават чрез шаблони, в които разработчикът може да укаже комбинация от HTML и код за свързване с данните. Примерно следният **ItemTemplate** ще покаже полето Name от източника на данни:

```
<asp:DataList runat="server" id="lstCharacterNames">
  <ItemTemplate>
    <%# DataBinder.Eval(Container.DataItem, "Name") %>
  </ItemTemplate>
</asp:DataList>
```

Можем лесно да разширим горния шаблон, за да покажем Name полето удебелено и под него да добавим поле ID:

```
<asp:DataList runat="server" id="lstCharacterNamesAndIDs">
  <ItemTemplate>
    <b><%# DataBinder.Eval(Container.DataItem, "Name") %></b>
    <br/>
    <%# DataBinder.Eval(Container.DataItem, "ID") %>
  </ItemTemplate>
</asp:DataList>
```

За всеки запис в източника на данни на **DataList**, се рендира изходен HTML след като се оцени свързването, указано в **ItemTemplate**. Поддържат се следните типове шаблони:

- **ItemTemplate** – шаблон за елемента
- **AlternatingItemTemplate** – ако е указан, всеки следващ елемент от източника на данни, използва този шаблон вместо **ItemTemplate**.
- **EditItemTemplate** – шаблон на елемента в режим на редакция.
- **HeaderTemplate** – шаблон за заглавния елемент. Показва се само ако свойството ShowHeader е true.
- **FooterTemplate** - шаблон за заключителния елемент. Показва се само ако свойството **ShowFooter** е **true**.
- **SelectedItemTemplate** – шаблон за избран елемент
- **SeparatorTemplate** – шаблон, прилаган след всяко добавяне на **DataListItem**.

По подразбиране, **DataList** показва всеки елемент като ред в HTML таблица. Чрез свойството **RepeatColumns** можем да укажем колко елементи искаме да се съдържат на всеки ред. Можем чрез свойството **RepeatLayout**, което приема стойности **Table** или **Flow**, да зададем да се ползват **** тагове вместо **<table>**, Тези допълнителни възможности правят **DataList** контролата по-гъвкава от **DataGrid**.

С шаблона **EditItemIndex** и събитията **EditCommand**, **UpdateCommand** и **CancelCommand**, контролата **DataList** също поддържа редактиране на място, но реализацията изисква повече програмиране от страна на разработчика. Още по-трудоемко е имплементирането на възможности за сортиране и страниране в **DataList** контрола.

Контролата Repeater

Контролата **Repeater** предлага максимална гъвкавост в рендирането на HTML. Тя е удачно решение, когато не искаме да използваме нито HTML **<table>**, нито серия от **** тагове.

Repeater предлага следните пет шаблона, чиято функция вече ни е добре позната:

- **AlternatingItemTemplate**
- **FooterTemplate**
- **HeaderTemplate**
- **ItemTemplate**
- **SeparatorTemplate**

HeaderTemplate и **FooterTemplate** указват HTML, който да се покаже съответно преди и след данните, свързани с контролата. **AlternatingItemTemplate** и **ItemTemplate** указват HTML кода и свързващия синтаксис за рендиране на елементите от източника на данни.

Нека свързваме данни за героите от книгата "Мечо Пух" с **Repeater** контрола и едно от полетата е Name. Ако искаме да покажем списък с имената им в несортиран списък можем да използваме следния синтаксис:

```
<asp:Repeater runat="server" id="rptCharacterNames">
  <HeaderTemplate>
    <ul>
  </HeaderTemplate>
  <ItemTemplate>
    <li><%# DataBinder.Eval(Container.DataItem, "Name") %></li>
  </ItemTemplate>
  <FooterTemplate>
    </ul>
  </FooterTemplate>
</asp:Repeater>
```

Тъй като **Repeater** не е наследник на **WebControl** и не предлага свойства за указване на стила на форматиране, то ако искаме да покажем имената на героите с удебелен шрифт, трябва в **ItemTemplate** да добавим HTML тага **<b>**:

```
<ItemTemplate>
  <li><b><%# DataBinder.Eval(Container.DataItem, "Name")
    %></b></li>
</ItemTemplate>
```

Тази особеност на **Repeater** контролата води понякога до по-тежки, а следователно и по-трудно четими шаблони. Също така ако се наложи да реализираме сортиране и страничен преглед, трябва да го реализираме от нулата.

Предимствата на **Repeater** са в нейната гъвкавост и добра производителност.

Управление на състоянието

Уеб страниците се прехвърлят чрез HTTP протокола. Те не запазват състоянието си, тъй като не знаят дали заявките идват от един и същ клиент. Страниците се създават наново при всяко обръщение към сървъра. Ако не се използваха допълнителни механизми за управление на състоянието (state management), възможностите на уеб приложенията биха били много ограничени.

В класическите ASP приложения този проблем се решава по няколко начина, чрез: бисквитки (cookies), параметризирани адреси (query string), и чрез ASP обектите за приложение (application) и за сесия (session). В ASP.NET всички тези техники са на наше разположение, като възможностите им са обогатени в много отношения.

Подходите за управление на състоянието в уеб приложенията се разделят на две категории – от страна на клиента (Client-side) и от страна на сървъра (Server-side). При управление на състоянието от страна на клиента, сървърът не пази информацията информация между заявките, а тя се съхранява на страницата или на компютъра на клиента.

Първо ще разгледаме Client-side техники – бисквитки, параметризирани адреси, скрити полета и ViewState. След това ще направим обзор на сървърните механизми за управление на състоянието на ниво приложение и ниво сесия.

Бисквитки (Cookies)

Бисквитката (cookie) е малко парче информация, изпратена от уеб сървъра до клиентски браузър. Браузърът по подразбиране съхранява получената бисквитка и от него се очаква да я изпраща обратно към сървъра при всяка следваща заявка. Информацията в нея може да е произволна, стига цялостният размер на бисквитката (информацията и мета данни за самата бисквитка) да не надвишава 4 KB. Нека да разгледаме някои от свойствата, с които се характеризират бисквитките.

Свойства на бисквитките

Ето някои от по-важните свойства на бисквитките:

- **Expires** - указва кога изтича валидността на бисквитката. Ако не се укаже, бисквитката се запазва само в паметта. Ако това свойство се зададе, бисквитката се записва на твърдия диск и се пази за времето, което е указано. Когато браузърът изпраща дадена бисквитка, той проверява дали нейната валидност не е изтекла и ако така, той не я изпраща към сървъра, а я изтрива. Не трябва да забравяме, че потребителят може да изтрие бисквитката по всяко време.
- **Domain** – областта от интернет адреси, на които може да се праща бисквитката. По подразбиране това е адресът, от който е дошла, но

може да се укаже и друго. Браузърът изпраща само бисквитките, предназначени за поискания интернет адрес.

- **Path** – път на адресите, до които може да се праща бисквитката. Бисквитката няма да се праща на адреси от по-високо ниво в дървото на директориите. Пример: ако пътят е **/sites/stefan**, тя няма да се прати на **/sites/dido,** нито на **/sites**, но ще се прати на **/sites/stefan/pics**. По подразбиране стойността на това свойство е виртуалната директория, от която първоначално е дошла бисквитката, но може и да се промени.

Механизъм на работа с бисквитки

За да разгледаме по-подробно механизмът на бисквитките, нека имаме примерна бисквитка с име UserID и стойност "StefanDobrev" – името на клиента. Нека датата на изтичане (свойството **Expires**) е 17-ти януари 2006 г., областта (свойството **Domain**) е **devbg.org**, а пътят (свойството **Path**) – главната виртуална директория.

Ето частта от HTTP хедъра, засягаща бисквитката, която ще се получи при клиента:

```
Set-Cookie: UserID=StefanDobrev; path=/; domain=devbg.org;
Expires=Saturday, 17-Jan-06 00.00.01 GMT
```

Ако клиентският браузър е Internet Explorer, папката, в която ще се съхрани бисквитката, е **\Documents and Settings\Username\Cookies**, а файлът ще е с име: **username@domainname.txt**. В случая, ако потребителят на системата е sdobrev, файлът ще има име **sdobrev@devbg.org[1].txt**. При всяка следваща заявка към този домейн и път, указан в бисквитката, браузърът е длъжен да изпрати съдържанието на бисквитката в HTTP хедъра, който също изпраща. В случая това ще е:

```
Cookie: UserID: StefanDobrev;
```

Това изпращане ще продължи, докато е валидна бисквитката. Трябва да се има предвид, че потребителят може да настрои браузъра си, така че да не приема бисквитки.

Като структура бисквитките представляват таблица от наредени тройки от типа адрес-име-стойност. Браузърът разпознава сървъра по неговия URL адрес и изпраща само тези бисквитките, предназначени за него.

Приложения на бисквитките

Както вече знаем, HTTP протоколът не може да запази състоянието на дадена заявка (той е stateless протокол). Бисквитките могат да се използват да разберем дали заявките идват от един и същи клиент.

Чрез механизма на бисквитките, сървърът може да следи потребителя и да му връща персонализирано съдържание, спрямо неговите нужди, изис-

квания и интереси. Към това приложение може да причислим и възможността за проследяване поведението на потребителя и изграждане на карта на най-често посещаваните от него страници.

Друго тяхно приложение е за автоматично влизане на потребителя в дадена уеб базирана система при неговото следващо идване. Това е възможно поради факта, че бисквитките могат да останат неограничено дълго време при клиентския браузър.

Бисквитките в .NET Framework

В .NET Framework има два класа, които предоставят достъп за работа с бисквитки.

System.Net.Cookie се използва при направата на клиентски приложения, като им предоставя функционалността да четат бисквитките, върнати от дадена уеб заявка.

System.Web.HttpCookie се използва в ASP.NET за достъп до бисквитките в уеб приложение. Чрез свойството **Cookies** на класовете **HttpResponse** и **HttpRequest** имаме достъп до колекция, която съдържа всички бисквитки, изпратени от сървъра или върнати от клиента.

Пример – четене от бисквитка

С този пример ще илюстрираме как може да се извлече дадена бисквитка от клиентска заявка и да се използва стойност, съхранена в нея:

```
HttpCookie cookie = Request.Cookies["UserID"];
if ( cookie != null )
{
   LabelUsername.Text = cookie["Username"];
   LabelExpires.Text = cookie.Expires;
}
```

Скрити полета

Скритите полета са подобни на текстовите полета, но с тази разлика, че не се показват в браузърите. Когато една страници е пратена до сървъра, съдържанието на скритите полета се праща в HTTP Form колекцията, заедно със стойностите на другите полета. Скритото поле играе ролята на държател за информация, специфична за страницата.

Скритите полета като HTML елементи

Скритите полета в HTML (**hidden**) имат следните атрибути:

- **name** - вътрешно име на полето, служещо за идентификация
- **value** - стойност, която да бъда изпратена до сървъра.

Ето един пример:

```
<input type="hidden" name="Language" value="English">
```

Скритите полета в .NET Framework

ASP.NET предоставя контролата **HtmlInputHidden**, която предлага функционалността на скрито поле:

```
protected System.Web.UI.HtmlControls.HtmlInputHidden Hidden1;

// Assign a value to Hidden field
Hidden1.Value = "invisible data";

// Petrieve a value
string str = Hidden1.Value;
```

Особености на скритите полета

За да използвате скритите полета, трябва да употребите HTTP POST метода за пращане на уеб страници.

Също така имайте предвид, че стойността не е напълно скрита за потребителя. Той може да я види в сорс кода на страницата и дори да я промени. Това прави скритите полета неудачни за съхраняване на чувствителна и конфиденциална информация.

Параметризирани адреси (Query Strings)

Параметризираните адреси предоставят лесен, но ограничен, начин за поддържане на информация за състоянието.

Пример за параметризиран адрес

Един параметризиран URL адрес може да изглежда по следния начин:

```
http://asp.net/getstarted/default.aspx?tabid=61
```

Когато се получи заявка за **getstarted/default.aspx**, можем от нея лесно да извлечем кой таб е бил избран чрез следния код:

```
string selectedTabID = Request.Params["tabid"];
```

Особености при използването на параметризирани адреси

Параметрите в заявката са видими в URL адреса и на практика не осигуряват никаква сигурност.

Повечето браузъри поддържат до 255 знака в URL. Това значително ограничава приложението на този подход.

Технологията ViewState

ViewState (визуално състояние) е технология, чрез която може да се съхрани информация за състоянието на сървърните контроли и данни, въведени от потребителя при последователни заявки към една и съща страница. Традиционните HTML елементи са без състояние и не запазват нито данните, нито настройките от страна на клиента.

Нека си представим една уеб форма, състояща се от много на брой полета, които клиентът трябва да попълни. След попълването ѝ трябва да валидираме въведената информация. Ако има неточности, потребителят е задължен отново да въведе цялата информация. Чрез технологията ViewState въведените от потребителя данни се запазват между заявките и не е нужно тяхното въвеждане да става отначало.

Сървърни контроли и ViewState

Благодарение на ViewState технологията, голяма част от сървърните контроли могат да запазват своето състояние (стойностите на отделните им свойства). Всяка динамична промяна на вътрешното състояние (промяна на свойство, свързване с данни и др.) на дадена сървърна контрола се запазва, за да може да бъде рендирана при клиента, когато има последователни заявки към една и съща страница.

ViewState като място за съхраняване на информацията

Освен за съхраняване вътрешното състояние на сървърните контроли ViewState може да се използва и за съхраняване на информация между няколко postback извиквания. Свойството **`ViewState`** на **`System.Web.UI.Control`** (базовия клас, който наследяват всички уеб контроли, включително и **`Page`**) предоставя достъп до речникова колекция от тип име-стойност, която може да се използва за съхраняване на информация.

Пример – съхраняване и извличане на данни от ViewState

Със следващия пример ще илюстрираме как може да съхраним информация във ViewState областта и след това да я извлечем от нея.

Запазване във ViewState:

```
ViewState["Username"] = TextBoxUsername.Text.Trim();
```

Извличане на вече съхранена информация от ViewState:

```
LabelUsername.Text = (string) ViewState["Username"];
```

Забележка: ако в речниковата колекция няма елемент със зададения ключ, се връща **`null`**.

Механизъм на работа на ViewState технологията

Всяка информация, добавена във ViewState (било то при динамична промяна на сървърна контрола или чрез свойството **ViewState**), се сериализира и се изпраща на клиента под формата на скрит HTML елемент със следния вид:

```
<input type="hidden" name="__VIEWSTATE"
 value="dDw5NjU1MTU1O3Q8cDxsPFRydWU7PjtsPFZpemliaWxpd
 Hk7Pj47Oz47Pm+DzsKPsEqi3imV9lUMfxhbK/Rc" />
```

Когато клиентът направи HTTP POST заявка към същата страница, съдържанието на скрития елемент се десериализира, възстановява се вътрешното състояние на сървърните контроли и се запълва речникова колекция, достъпна чрез **ViewState** свойството.

Сериализацията и десериализацията се извършват с помощта на класа **LosFormatter**, който е оптимизиран за работа с примитивни типове, символни низове, масиви и хеш-таблици.

Както вече отбелязахме, информацията, запазена във ViewState областта, се сериализира. Това означава, че ако искаме да запазим инстанция на дефиниран от нас потребителски клас, той трябва да е маркиран с атрибута **[Serializable]**.

Стойността на скрития елемент **__VIEWSTATE** е BASE64 представяне на сериализираните контроли от формата. Въпреки че тази информация не е лесно четима, тя не е криптирана и може да бъде декодирана.

Не съхранявайте конфиденциална информация във ViewState!

За да се избегне фалшификация на ViewState информацията, всеки път, когато ASP.NET създава сериализирания ViewState, автоматично към него се добавя и хеш кодът му. При следваща заявка се проверява дали данните от ViewState имат същия хеш код (т.е. дали не са променени). Тази опция може да се изключи, като в директивата **@Page** зададем стойност **false** на атрибута **EnableViewStateMac**.

Поддържане на ViewState

Ако дадена уеб страница съдържа множество контроли, цялостният размер на ViewState областта може да нарасне драстично, което от своя страна увеличава размера на страницата, която се изпраща към клиента. В подобни случаи може да ограничим използването на ViewState само върху контролите, които се нуждаят от него. Например за контрола от тип **Label**, която има зададено свойство **Text** в **aspx** страницата и знаем, че това и останалите й свойства няма да се променят, е разумно да изключим ViewState-а. Това може да стане така:

```
<asp:Label ID="LabelName" Runat="server" Text="Stefan"
  EnableViewState="False" />
```

Изключването на ViewState може да стане и на ниво страница:

```
<%@ Page EnableViewState="False" %>
```

Това е удобно, ако искаме да разрешим използването на ViewState само за определени контроли.

Запазване на ViewState в Session обекта

Когато обемът на информацията, съхранена във ViewState, нарасне, цялостният размер на HTML страницата, изпратена към клиента, също нараства. Това може да доведе до изпращане на страници с размер от около 0.5 – 1 MB, което не е препоръчително.

Един сценарий, в който е удачно да поддържаме малък ViewState е, когато разработваме приложения за мобилни клиенти. В тези ситуации е добре да го съхраняваме на друго място, като избегнем неговото рендиране при клиента и в същото време запазим състоянието. В примера ще покажем как това може да стане в **`Session`** обекта. За целта ще препокрием два от виртуалните методи на класа **`System.Web.UI.Page`**: **`LoadPageStateFromPersistenceMedium()`** и **`SavePageStateToPersistenceMedium(…)`**. Ето и кода, който осъществява това:

```
protected override object LoadPageStateFromPersistenceMedium()
{
   return Session["ViewState"];
}

protected override void SavePageStateToPersistenceMedium(
  object viewState)
{
   Session["ViewState"] = viewState;
}
```

С този пример демонстрирахме как можем да контролираме механизма на записване и зареждане на ViewState информацията. Може да се използват и произволни други места за съхранение: бази от данни, файлове, собствени скрити полета и др.

Състояние на приложението

В рамките на ASP.NET приложение информация да бъде споделяна чрез класа **`HttpApplicationState`** (достъпван най-често чрез **`Application`** свойството на **`HttpContext`** обекта). Този клас ни предоставя речникова колекция, където можем да съхраняваме обекти и скаларни стойности, свързани с множество уеб заявки и клиенти.

При първата заявка към URL ресурс от виртуалната директория на ASP.NET приложение се създава инстанция на класа **`HttpApplicationState`**. По време на всяка заявка всички модули **`HttpModule`** и обработчици **`HttpHandlers`** (в това число ASP.NET страниците), имат достъп тази инстанция чрез свойството **`Application`** на **`HttpContext`** обекта.

За поддръжка на състояние на ниво приложение ASP.NET ни предоставя:

- Речникова колекция, достъпна за всички обработчици на заявки в приложението.
- Лесен механизъм за синхронизация до променливите на състоянието.
- Сигурност, че други ASP.NET приложения не могат да достъпват и променят състоянието на нашето приложението.

Използване на състоянието на приложението

Променливите на състоянието на **`Application`** обекта, са на практика глобални за ASP.NET приложение. Затова при вземане на решение дали да ги използваме, трябва да имаме предвид следните фактори:

- Памет - паметта не се освобождава докато променливата не бъде заменена или премахната. В някои случаи е лоша идея да се държат постоянно в паметта рядко достъпвани данни с голям размер.
- Нишкова безопасност – ако обектите, които съхраняваме, не са нишково обезопасени, то трябва да положим допълнителни усилия за синхронизиране на достъпа до тях.
- Скалируемост – при използване на заключвания за осигуряване на нишкова безопасност, операционната система блокира другите работещите нишки, чакащи за ресурса. Това може да доведе до значително падане на производителността на приложението.
- Възстановяване на данните – по време на изпълнение на приложението, домейнът на приложението може да бъде унищожен във всеки момент (в резултат на срив, промени в кода, планирано рестартиране на процеса, и др.). В такъв случай данните за състоянието на приложението ще бъдат загубени. Ако такова поведение е нежелателно, трябва да предприемем допълнително стъпки за решаване на проблема.
- Състоянието на приложението не е споделено в рамките на уеб ферма (приложение, изпълнявано на няколко сървъра) или уеб градина (приложение, изпълнявано на няколко процеса на един сървър). Променливите, съхранявани в състоянието на приложението, са глобални само в рамките на един процес.

Въпреки тези особености, променливите на ниво приложение могат да бъдат много полезни. В тях можем да пазим рядко извличана, но често

достъпвана информация от бази от данни и така да подобрим значително скоростта на обработка на заявките. От друга страна, този ефект можем да бъде постигнат и с механизмите за кеширане в ASP.NET, които ще разгледаме по-късно.

Колекции за състоянието на приложението

Класът **HttpApplicationState** предоставя две колекции: **Contents** и **StaticObjects**.

Колекцията **Contents** дава достъп до променливите добавени по следния начин:

```
Application["AppStartTime"] = DateTime.Now;
```

Можем изрично да използваме свойството **Contents**:

```
Application.Contents["AppStartTime"] = DateTime.Now;
```

Колекцията **StaticObjects** предоставя достъп до променливите, дефинирани чрез **<object runat="server">** тагове във файла **Global.asax**:

```
<object runat="server" scope="application"
   ID="MyInfo" PROGID="MSWC.MYINFO">
</object>
```

Можем да използваме така дефинираните обекти по следния начин:

```
<html>
   </body>
      Application Level Title: <%= MyInfo.Title %>
   <body>
</html>
```

Синхронизация на достъпа до състоянието на приложението

Няколко нишки на приложението може едновременно да достъпват стойности, съхранени в състоянието на приложението. Следователно трябва да се погрижим да осигурим, че използваме тези променливи по безопасен начин.

За целта, класът **HttpApplicationState** предоставя два метода **Lock()** и **Unlock()**, които ограничават достъпа до променливите на приложението само до една нишка.

Извикване на **Lock()** метода на **Application** обекта кара ASP.NET да блокира опитите на нишките да достъпват състоянието на приложението, до извикване на **Unlock()**.

Следният код демонстрира техниката на заключване:

```
Application.Lock();
Application["SomeGlobalCounter"] =
    (int)Application["SomeGlobalCounter"] + 1;
Application.UnLock();
```

Ако не извикаме **`Unlock()`**, то заключването ще бъде премахнато автоматично щом приключи заявката, или при timeout, или при появяване на необработено изключение, което да прекрати обработката на отговора.

Състояние на сесиите

ASP.NET предоставя възможност за запазване на информация за взаимодействието с определен потребител между отделните заявки. При разработката на уеб приложения често се налага да реализираме такава функционалност. Типичен пример е уеб-базирана система за работа с електронна поща. При нея потребителите първо се идентифицират чрез потребителско име и парола, а след това системата ги "познава" до момента на затварянето на уеб браузъра или излизане от системата.

Вградените в ASP.NET възможности за поддръжка на потребителска сесия (session state) ни позволяват да:

- Идентифицираме и класифицираме автоматично в логическа сесия всички заявки, идващи от един и същ браузър.
- Запазваме данни на сървъра за сесията, с цел използването им между множество отделни заявки.
- Да обработваме в кода ни събития, свързани със сесията (**`Session_OnStart`**, **`Session_OnEnd`**, и т.н.).
- Автоматично да бъдат освобождаване данните за сесията, ако в определен период от време не се получи заявка от браузъра.

Поддръжката на сесии в ASP.NET се характеризира с:

- Леснота за ползване.
- Надеждно запазване на данни, устойчиво на рестартиране на IIS или на работния процес на ASP.NET.
- Скалируемост в уеб ферма и уеб градина.
- Възможност за функциониране и без HTTP бисквитки.
- По-добро бързодействие спрямо класическото ASP.

Забележка: Състоянието на сесиите не се запазва извън границите на едно уеб приложение.

Идентифициране на сесия

Всяка активна ASP.NET сесия се идентифицира и проследява чрез 120-битов низ **SessionID**. Той е съставен от ASCII символи и може да участва в URL адреси.

SessionID стойностите се генерират така, че да са уникални и достатъчно произволни, за да не може по идентификатор на нова сесия да се открие идентификатор на предишна.

SessionID низовете се прехвърлят между заявките чрез HTTP бисквитка или чрез включване в URL адресите, в зависимост от настройките на приложението.

Запазване на състоянието в рамките на сесия

ASP.NET позволява запазването на произволни данни за сесия в речникова колекция, която се съхранява в паметта на IIS процеса.

Когато използваме режим in-process, т.е. данните за сесията се пазят директно в ASP.NET процеса, трябва да имаме предвид, че те ще бъдат загубени ако **aspnet_wp.exe** или application domain бъдат рестартирани. Това може да случи в следните сценарии:

- Наличие на атрибут в елемента **<processModel>** на **Web.config** файла, който да доведе до стартиране на нов ASP.NET работен процес (примерно указан лимит на паметта).
- Промяна в **Global.asax** или **Web.config** файловете.
- Промени в **\Bin** директорията на уеб приложението.

В режим out-of-proc, вместо да се поддържат живи обекти в работния процес, т.нар. State Server съхранява състоянието в паметта, а работният процес се обръща при нужда към него. В режим SQL, състоянието на сесията се пази в SQL Server.

ASP.NET работният процес сериализира сесийните обектите в края на всяка заявка. При следваща заявка данните се извличат от State сървъра като двоични потоци, десериализират се и се поставят в нова колекция, вече готови за употреба. По този начин може да се реализира запазване на състоянието при срив на работния процес. А допълнително може паралелно да работят няколко процеса, което прави този подход по-скалируем.

Структура на състоянието на сесия

Класът, имплементиращ поддръжката на сесийни данни е HTTP модула **SessionStateModule**. Той генерира и извлича уникални идентификатори на сесията и си взаимодейства с доставчика на услуга по съхранение на данните за сесията.

Подобно на **HttpApplicationState**, класът **SessionState** предоставя две колекции **Contents** и **StaticObjects**. Работата с тях е аналогична на тази на **HttpApplicationState**. Ще дадем кратки примери:

```
Session["AppStartTime"] = DateTime.Now;
Session.Contents["AppStartTime"] = DateTime.Now;
```

Ето как указваме обхват "**Session**" на променлива в **Global.asax** файла:

```
<OBJECT RUNAT="SERVER" SCOPE="SESSION"
   ID="MyInfo" PROGID="Scripting.Dictionary">
</OBJECT>
```

Конфигуриране на състоянието на сесия

Както споменахме, в ASP.NET може да избираме между три режима на съхраняване на данни за сесии: in-process, State Server, и SQL Server. Независимо на кой механизъм се спрем, конфигурирането на състоянието на сесиите протича в две фази. Първо, модулът за състояние се добавя към HTTP заявката. По подразбиране, тази настройка се задава на ниво компютър във файла **Machine.config**. Ето примерна секция от този файл:

```
<httpmodules>
    <add name="sessionState"
type="System.Web.SessionState.SessionStateModule,
Version=1.0.3300.0,Culture=neutral,
PublicKeyToken=b77a5c561934e089" />
</httpmodules>
```

Втората стъпка е да укажем желаните атрибути за конфигурация чрез атрибута **<sessionState>**. Сред по-важните настройки са:

- **mode** - режим ("Inproc", "StateServer" или "SQLServer").
- **cookieless** – дали да се използват бисквитки.
- **timeout** – таймаут за изтичане на сесия.

Валидация на данни

Всяко едно интерактивно приложение, позволява на потребителите си да въвеждат данни. Често се случва по невнимание или преднамерено да бъдат въведени грешни данни. Ако не се обработят такива неочаквани ситуации, те могат да доведат до неочаквано поведение на приложението и дори до сриване на цялата система. Когато се очаква въведените данни да са от определен тип (примерно целочислен, реален и т.н.), в определен интервал (примерно от 0 до 100) или да отговарят на по-сложни правила (примерно да са валиден e-mail), разработчикът е длъжен да подсигури, че въведените от потребителя данни отговарят на тези изисквания.

Процесът на проверка на въведените данни наричаме валидация на данните.

Уеб приложенията, разработени с ASP.NET уеб форми, предоставят възможност за интерактивна работа на потребителя. Въведените от потребителя данни се проверяват и ако не отговарят на очакваните от приложението, уеб формата не позволява преминаване към друга форма, докато данните не бъдат коригирани.

За да се реализира ефективна валидация на данните, ASP.NET ни предоставя набор от контроли наречени валидатори. Те в значителна степен улесняват извършването на проверките.

RequiredFieldValidator – проверка за наличие на данни

Контролата **RequiredFieldValidator** проверява дали потребителят е въвел изобщо някакви данни. Това е един от най-често използваните в практиката валидатори.

RequiredFieldValidator – пример

Проверка дали потребителят е попълнил полето за име в дадена форма:

```
<asp:RequiredFieldValidator id="requiredFieldValidator"
  runat="server" ErrorMessage="Name Field is required"
  ControlToValidate="txtName">*</asp:RequiredFieldValidator>
```

Понякога ни се налага да проверим дали данните, въведени от потребителя, са различни от някаква първоначална стойност. Примерно ако използваме контролата **DropDownList**, в която са изброени всички държави, и като първи негов елемент стои "Select your country", естествено е да не искаме да позволим на потребителя да избере служебния запис. Тук пак може да използваме **RequiredFieldValidator**, като само трябва да укажем първоначална стойност в свойството **InitialValue**.

RequiredFieldValidator – още един пример

Ето как се използва валидаторът в този случай:

```
<asp:DropDownList id="ddlCountries" runat="server">
  <asp:ListItem Value="0">Select your country</asp:ListItem>
  <asp:ListItem Value="1">Bulgaria</asp:ListItem>
  <asp:ListItem Value="2">USA</asp:ListItem>
  <asp:ListItem Value="3">United Kingdom</asp:ListItem>
</asp:DropDownList>

<asp:RequiredFieldValidator id="rfvCountry" runat="server"
  ErrorMessage="Please select your country"
  InitialValue="0" ControlToValidate="ddlCountries"> *
```

```
</asp:RequiredFieldValidator>
```

CompareValidator – проверка на входните данни

CompareValidator е валидатор, полезен в случаите, когато искаме:

- да сравним входните данни на една контрола с тези на друг;
- да сравним входните данни с константна стойност
- да установим дали въведените данни са от определен тип.

Ако искаме да сравним данните в две контроли (примерно две **TextBox** контроли), първо трябва да укажем на валидатора коя е базовата контрола (**ControlToCompare)** и коя е валидираната (**ControlToValidate**). За да се определи какво точно сравнение да се извърши (равенство, по-голямо, по-малко, по-голямо или равно, по-малко или равно или различно), се използва свойството **Operator**, което има стойности – **Equal**, **GreaterThan**, **GreaterThanEqual**, **LessThan**, **LessThanEqual**, **NotEqual**. Допълнително чрез свойството **Type** на валидатора, може да се укаже и типът на данните, които очакваме да бъдат попълнени от потребителя. Валидните типове са **String**, **Integer**, **Double**, **Date**, **Currency**. Стойността по подразбиране е **String**. Ако сме задали друга, преди да се извърши сравнение между стойностите на зададените контроли, се проверява дали стойността на контролата **ControlToValidate** е от съответния тип. В случай че не е, проверката за валидност връща отрицателен резултат. Ако стойността на контролата **ControlToCompare**, не е от зададения тип, а тази в **ControlToValidate** е очакваната, то проверката минава. Затова трябва да се прави изрична валидация за типа на данните в контролата **ControlToCompare**. Друга особеност на **CompareValidator** е, че ако няма въведена стойност в една от двете контроли **ControlToValidate** и **ControlToCompare**, проверката минава. Поради това се налага винаги да се използва в комбинация с **RequiredFieldValidator**.

В случай, че не е зададено свойството **ControlToCompare**, а **ValueToCompare**, сравнението е със стойността във **ValueToCompare**. Ако тя не е от указания тип в свойството **Type**, се хвърля **HttpException**. В случай, че са зададени и **ControlToCompare** и **ValueToCompare**, с приоритет е **ControlToCompare**.

Ако искаме само да проверим дали стойността на контролата **ControlToValidate** е от зададения тип, задаваме на свойството **Operator**, стойност **DataTypeCheck**.

CompareValidator – примери

Проверка дали стойностите на две текстови полета съдържат еднакви стойности:

```
<asp:CompareValidator id="compareValidator" runat="server"
    ErrorMessage="The two fields do not match"
    ControlToValidate="TextBox1" ControlToCompare="TextBox2"
    Type="String" Operator="Equal"> </asp:CompareValidator>
```

Проверка дали стойността на едно текстово съвпада с низа "Бай Киро":

```
<asp:CompareValidator id="compareValidator" runat="server"
    ErrorMessage="The value is not 'Бай Киро'"
      ControlToValidate="TextBox1" ValueToCompare="Бай Киро"
      Type="String" Operator="Equal"> </asp:CompareValidator>
```

Проверка дали стойността на едно поле е от целочислен тип:

```
<asp:CompareValidator id="compareValidator" runat="server"
    ErrorMessage="You have to enter an integer value"
    ControlToValidate="TextBox1" Type="Integer"
    Operator="DataTypeCheck" /> </asp:CompareValidator>
```

RangeValidator – проверка попадане в интервал

Често на уеб разработчиците им се налага да осигурят, че данните, въведени от потребителя, са в определен интервал (примерно, че попадат в период ограничен от две дати или са между две числени стойности). За тази цел може да използваме контролата **RangeValidator**, която има много общи свойства с **CompareValidator**. Разликата е, че при нея трябва да укажем граници за стойностите в контролата: минимална - **MinimumValue** и максимална – **MaximumValue**. Както при **CompareValidator**, ако има несъответствие на въведените данни и свойството **Type**, възниква **HttpException**. Ако в контролата **ControlToValidate** не са въведени данни, проверката минава успешно. Затова често се налага да се комбинира с **RequiredFieldValidator**.

RangeValidator – пример

Проверка за дата от 2006 година:

```
<asp:RangeValidator
    id="rangeValDate"
    Type="Date"
    ControlToValidate="txtDate"
    MaximumValue="2006/12/31"
    MinimumValue="2006/1/1"
    runat="server"/>
```

RegularExpressionValidator – сравняване с регулярен израз

Валидация на входни данни чрез регулярни изрази е описана в детайли в едноименната тема. ASP.NET предоставя контрола **RegularExpression Validator** за позитивна валидация на въведените от потребителя данни. Трябва да се укажат контролата, която ще бъде валидирана - **ControlToValidate**, и регулярният израз, с който да се извърши проверката - **ValidationExpression**. Валидацията може да се извърши както при клиента, така и на сървъра. При проверка при клиента, се използват регулярните изрази в JavaScript, чиито синтаксис е подмножество на синтаксиса, поддържан от класа **Regex**. Препоръчително е при задаване на **ValidationExpression** да се използва синтаксиса на **JScript** регулярните изрази, за да се избегнат несъответствия. Както при предходно изброените валидатори, така и при този, ако в контролата **ControlToValidate**, не са въведени данни, проверката минава успешно.

RegularExpressionValidator – пример

Проверка на валиден e-mail адрес (това е опростен пример, регулярният израз, който е обхваща всички валидни адреси е значително по-голям):

```
<asp:RegularExpressionValidator id="revEmail" runat="server"
  ErrorMessage="Email is not valid."
  ControlToValidate="txtEmail"
  ValidationExpression=
      "\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*"> !
          </asp:RegularExpressionValidator>
```

CustomValidator – произволна проверка

Описаните по-горе контроли покриват голяма част от реалните нужди на разработчика за валидация на данните. Когато са налага прилагане на по-сложна логика за валидиране на данните можем да използваме контролата **CustomValidator**.

Важна характеристика на валидаторите е, че валидацията винаги се извършва и на сървъра, дори да се е извършила на клиента. В досега изброените до момента контроли, това става автоматично, но при **CustomValidator** се налага сами да добавим тези функции. Това става с прихващане в уеб формата на събитието **ServerValidate**, което е с аргумент обект от тип **ServerValidateEventArgs**. За целта са ни нужни стойността на контролата, която се валидира, и променлива, в която да върнем резултата. Аргументът от тип **ServerValidateEventArgs**, подаден от събитието, съдържа стойността на валидираната контрола **ControlToValidate** в свойството **Value**. След извършване на проверката, трябва да върнем резултат чрез свойството **IsValid**.

CustomValidator – пример

За да илюстрираме употребата на **CustomValidator** контрола, нека разгледаме форма със следното съдържание: два бутона за алтернативен избор (**RadioButton**), чрез които потребителят указва своя пол (мъж / жена), и едно текстово поле, в което той попълва своя ЕГН. ЕГН е десетцифрен номер, в който първите шест цифри са за рождената дата, седмата и осмата са за служебна информация за район, деветата е за пола, а десетата цифра е контролна. Алгоритъмът на пресмятане на десетата цифра е известен, но сега няма да го дискутираме.] Искаме да сверим дали дадено ЕГН, съответства на избрания пол. Ако един човек е мъж, деветата цифра е четна, ако е жена – нечетна. На тази база ще изградим валидация чрез контролата **CustomValidator**. Частта от уеб формата, която ни интересува, е:

```
<asp:RadioButton ID="rbtnFemale" Runat="server"
   Text="Жена" Checked="True" GroupName="Gender" />
<asp:RadioButton ID="rbtnMale" Runat="server"
   Text="Мъж" GroupName="Gender" />
<asp:TextBox id="txtEGN" runat="server" />
<asp:CustomValidator id="cvEGN" Runat="server"
   ErrorMessage="Въвели сте невалиден ЕГН"
   EnableClientScript="True"
   ControlToValidate="txtEGN"
   ClientValidationFunction="ValidateEGN">*</asp:CustomValidator
>
```

Освен стандартния атрибут **ControlToValidate** на **CustomValidator** контролата сме задали и атрибута **ClientValidationFunction**. В него се задава името на JavaScript функцията, отговаряща за валидацията при клиента. В кода зад формата се абонираме за събитието **ServerValidate**:

```
cvEGN.ServerValidate +=
   new ServerValidateEventHandler(cvEGN_ServerValidate);
```

Функцията **cvEGN_ServerValidate(…)** реализира валидацията на сървъра:

```
private void cvEGN_ServerValidate(object source,
   ServerValidateEventArgs args)
{
   string pattern = @"^[0-9]{10}$";
   if( Regex.IsMatch(args.Value, pattern) )
   {
      int genderDigit =
                  Convert.ToInt32( args.Value.Substring(8,1) );
      if( (genderDigit % 2) == 0 )
      {
         if( rbtnMale.Checked )
         {
```

```
            args.IsValid =true;
         }
         else
         {
            args.IsValid = false;
         }
      }
      else
      {
         if( rbtnFemale.Checked )
         {
            args.IsValid =true;
         }
         else
         {
            args.IsValid = false;
         }
      }
   }
   else
   {
      args.IsValid = false;
   }
}
```

Тази функция използва елементарен регулярен израз, за да провери дали потребителят е въвел смислени данни. Функцията, която отговаря за валидацията при клиента, се реализира на език, поддържан от браузърите (най-често се използва JavaScript и VBScript). Ето функцията **`ValidateEGN(…)`** на JavaScript:

```
<script language="javascript">
   function ValidateEGN( source, arguments )
   {
      var pattern = /^[0-9]{10}$/;
      var rbtnMale = document.getElementById("rbtnMale");
      var rbtnFemale = document.getElementById("rbtnFemale");

      if ( pattern.test(arguments.Value) )
      {
         var genderDigit = arguments.Value.substr(8,1) ;
         if( (genderDigit % 2) == 0 )
         {
            if( rbtnMale.checked )
            {
               arguments.IsValid =true;
            }
            else
            {
               arguments.IsValid = false;
```

```
            }
         }
         else
         {
            if( rbtnFemale.checked )
            {
               arguments.IsValid =true;
            }
            else
            {
               arguments.IsValid = false;
            }
         }
      }
      else
      {
         arguments.IsValid = false;
      }
   }
</script>
```

ValidationSummary – списък на грешките

Когато потребителите попълват форми, могат да въведат грешни данни в повече от една контрола. В такива случаи е най-удобно да се изкара списък (резюме) на грешките и за целта ASP.NET ни предоставя контрола **ValidationSummary**. При извършване на валидация всеки валидатор проверява дали са въведени коректни данни и ако не са, в контрола **ValidationSummary** се извеждат съобщенията за грешка, зададени чрез атрибута **ErrorMessage**.

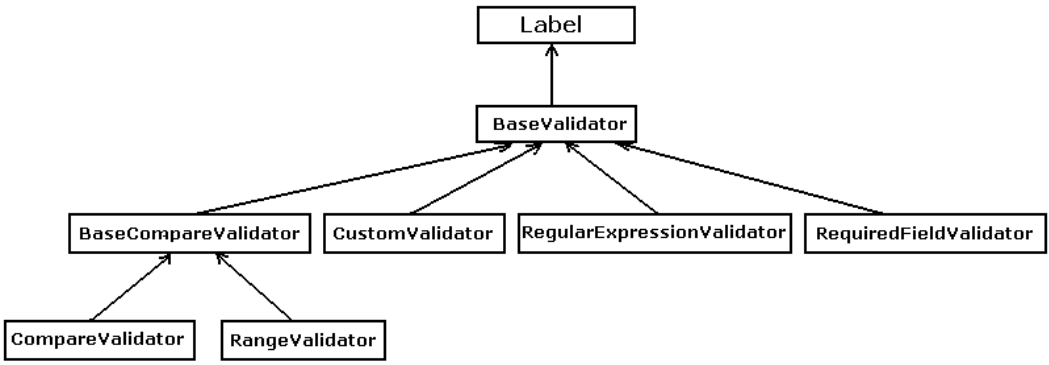
Контролата **ValidationSummary** предлага следните опции:

- Свойството **DisplayMode** определя по какъв начин да се покажат грешките. Възможните стойности за него са – **BulletList**, **List**, **SingleParagraph**. Стойността по подразбиране е **BulletList**.
- Свойството **EnableClientScript** определя дали при клиента да се изпълни скрипт и да се избегне ходене до сървъра, или списъкът с грешките да се попълни чак на сървъра. Стойността по подразбиране е **true**.
- Свойството **ShowSummary** определя дали списъкът с грешки да се показва на потребителя. Стойността по подразбиране е **true**.
- Свойството **ShowMessageBox** определя дали да се покаже на потребителя списъкът с грешки под формата на **MessageBox**. Ако стойността на този атрибут е **true**, за да се покаже **MessageBox**, е необходимо и стойността на **EnableClientScript** да е **true**. Стойността по подразбиране е **false**.

- Свойството **HeaderText** определя какво да е заглавието на резюмето с грешки. Стойността по подразбиране е празният низ.

Йерархия на класовете валидатори

Следната клас диаграма описва йерархията на валидаторите:



Валидаторите се явяват специализирани **Label** контроли. Базовият клас **BaseValidator** дефинира общите за всички валидатори свойства – **ControlToValidate**, **Display**, **EnableClientScript**, **Enabled**, **ErrorMessage**, **IsValid**. **RangeValidator** и **CompareValidator** наследяват от **BaseCompareValidator** общото свойство **Type**.

Общи свойства за валидаторите

Като наследници на базовия клас **BaseValidator**, всички валидатори имат някои общи свойства:

- **ControlToValidate** – задава на коя контрола да бъдат проверени входните данни.
- **Display** – контролира по какъв начин да се показва текста на валидатора (става дума за стойността на свойството **Text**, а не за стойността на свойството **ErrorMessage**). Възможните стойности за този атрибут са – **Dynamic**, **Static**, **None**. Стойността по подразбиране е **Static**. При **None** не се показва нищо. Разликата между **Dynamic** и **Static** е малка и е свързана с факта, че валидаторите се визуализират (render) като **** тагове. Когато **Display** има стойност **Dynamic**, атрибутът **style** на **** тага изглежда така: **style="color:Red; display:none;"**. Докато при **Static**, атрибутът **style** на **** тага изглежда така: **style="color:Red; visibility:hidden;"**. Разликата е в това, че пространството, заето от текста на валидатора при стойност **Static**, е предварително заделено, докато при **Dynamic** се заделя при появата на текста.
- **EnableClientScript** – указва дали валидацията за дадения валидатор ще се извърши и при клиента, или само на сървъра.

Приема стойности **true** и **false** (по подразбиране - **true**). Всеки един от валидаторите с изключение на **CustomValidator** има реализация на проверката за валидност при клиента. Ако стойността на **EnableClientScript** е **true**, при клиента се прави проверка, като в случай на невалидни данни се спестява ходенето до сървъра.

- **Enabled** – контролира дали съответният валидатор е активен или не. Стойността по подразбиране е **true**.
- **ErrorMessage** – съхранява съобщението за грешка, което се показва на потребителя при въведени некоректни данни.
- **IsValid** – показва дали съответният валидатор е минал успешно проверката на данните. Стойността на това поле по подразбиране е **true**.

Кога и къде се извършва валидацията?

Когато използваме стандартните валидатори на ASP.NET, проверката за валидност се извършва винаги на сървъра. В зависимост от стойността на полето **EnableClientScript** може да има проверка и при клиента, но задължително се проверява и на сървъра.

При проверката за валидността на данните, се задава булева стойност на свойството **IsValid** на уеб формата, в зависимост дали проверката е минала успешно или не. Тази стойност се задава автоматично от метода на формата **Validate()**, който се извиква по време на изпълнението на уеб формата. За да разчитаме, че полето **IsValid** съдържа коректна стойност, трябва да знаем в кой етап от модела на изпълнение на формата се извиква този метод. Честа грешка в практиката е да се проверява дали формата е валидна при събитието **Load** на формата. Това е погрешно, защото методът **Validate()** се извиква след събитието **Load** и преди събитията, свързани с останалите контроли на формата (**Click**, **SelectionChange** и други).

Има случаи, в които не искаме да се проверяват за валидност въведените от потребителя данни. Най-тривиалният пример е с форма, в която потребителят трябва да въведе някакви данни и да потвърди с бутона **Submit**, но да може и да се откаже с бутона **Cancel**. В този случай трябва при натискане на **Submit** да се извърши валидация, а при натискане на **Cancel** да не се прави такава. За целта на атрибута **CausesValidation** на бутона **Cancel** се задава стойност **false**.

Защо винаги на сървъра?

Валидацията при клиента става чрез скриптове, изпълнявани на машината на потребителя. Потребителите могат да вдигнат нивото на сигурност и изцяло да забранят изпълнението на скриптове.

Скриптовете за валидация са базирани на така наречения Document Object Model. При различните браузъри (Internet Explorer, Firefox, Netscape, Opera…) и дори при различните версии на един браузър този модел е реализиран по различен начин, въпреки утвърдените общи стандарти. В резултат на това валидацията при различните браузъри може да даде различни резултати.

Допълнително трябва да се има предвид, че скриптовете са просто обикновен текст, интерпретиран от браузъра на клиента. Потребителят има пълната свобода да редактира скрипта и да го накара да прави това, което той пожелае.

Тези факти около сигурността и консистентността на скриптовете са причината проверката за валидност на данните винаги да се прави на сървъра.

Особености при валидацията при клиента

Валидацията при ASP.NET 1.1 има някои особености. Тя има едно важно ограничение:

Валидацията, реализирана чрез стандартните валидатори на ASP.NET 1.1, работи само с Internet Explorer.

За да извърши валидация, ASP.NET рендира допълнителен скрипт на JavaScript, който е съвместим само с Internet Explorer. За да се извърши проверка за всички валидатори при клиента, те се поставят в JavaScript масив и след това един по един проверяват дали потребителят е въвел коректни данни. Частта от скрипта, която се поддържа само от Internet Explorer е:

```
document.all["validator_name"]
```

Така, ако уеб формата има четири **`RequiredFieldValidator`** контроли, JavaScript кодът изглежда по следния начин:

```
var Page_Validators = new Array(
  document.all["RequiredFieldValidator1"],
  document.all["RequiredFieldValidator2"],
  document.all["RequiredFieldValidator3"],
  document.all["RequiredFieldValidator4"]);
```

Проблемът може да бъде избегнат и в ASP.NET 1.1, но за целта трябва програмиста да реализира свои контроли за валидатори. В следващата версия на ASP.NET (2.0) този проблем е решен и валидацията при клиента работи с всички уеб браузъри.

Потребителски контроли

HTML и уеб сървър контролите предлагат лесен начин за повторно използване (reuse) на функционалност. Но често се налага на няколко места да искаме да използваме комбинация от група контроли, които да имат еднакъв вид и/или поведение. За целта ASP.NET предлага възможност за разработка на потребителски контроли (user controls). Те предоставят удобен начин за споделяне на функционалност и потребителски интерфейс между страниците на приложението.

Потребителски контроли и уеб форми

Потребителската контрола е елемент подобен на ASP.NET уеб форма, който може да се вгражда в други ASP.NET уеб форми. Подобно на уеб формите, потребителските контроли са сървърни компоненти, които предлагат потребителски интерфейс и функционалност.

Основната разлика между потребителските контроли и уеб страниците е, че първите не са предназначени да се показват директно в браузър. За да бъдат използвани, трябва да бъдат включени в уеб форма.

Потребителските контроли са наследници на **`System.Web.UI.UserControl`** в обектния модел на ASP.NET. Те се описват във файл с разширение (**`.ascx`**).

Предимства при използването на потребителски контроли

- Самостоятелни – потребителските контроли са самостоятелни и предоставят отделни пространства от имена (namespaces) за променливите. Така не се получават колизии със съществуващи методи и свойства на страницата, която ползва потребителската контрола.
- Преизползваеми (reusable) – потребителските контроли могат да се използват повече от веднъж на една или няколко страници.
- Езиково неутрални – потребителските контроли могат да бъдат писани на различен програмен език от използвания в страницата, в която се разполагат.

Споделяне на потребителски контроли

Потребителските контроли могат да се споделят между всички страници на уеб приложението, но много трудно се споделят между различни уеб приложения. Ако искаме по-широко преизползване без copy&paste, трябва да разработваме Web custom контроли, чието създаването е много по-трудоемко.

Използване на потребителски контроли

Потребителската контрола може да се постави във всяка ASP.NET уеб форма. Формата, която добавя контролата, се нарича домакин **(host)**. Формата добавя контролата, като използва директивата **@Register**.

Примерно използване:

```
<%@ Register TagPrefix="demo" TagName="validNum"
Src="numberbox.ascx" %>
```

Атрибутът **TagPrefix** указва уникално пространство от имена за потребителската контрола, за да няма колизии, ако същата контрола се използва повторно. Атрибутът **TagName** е име на инстанцията на контролата. Атрибутът **Src** е релативен път до файла на контролата.

Създаване на потребителска контрола – пример

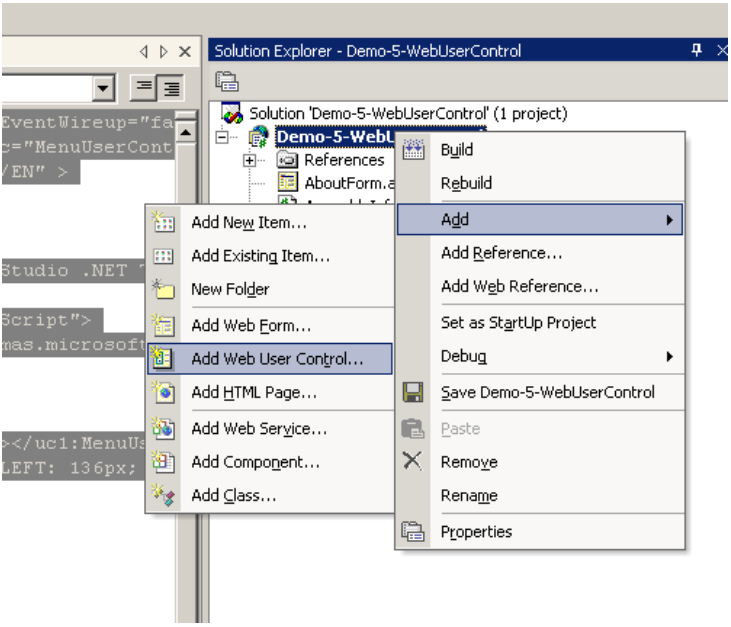
В този пример ще създадем потребителска контрола, която служи за меню. Менюто в един сайт би трябвало да присъства на всяка страница от сайта и затова е подходящо да го направим потребителска контрола. Така на всяка страница ще добавяме само меню контролата, вместо да създаваме меню от нулата.

Нека първо създадем три уеб форми – за начална страница (Main), за страница с контакти (Contacts) и за страница с информация (About).

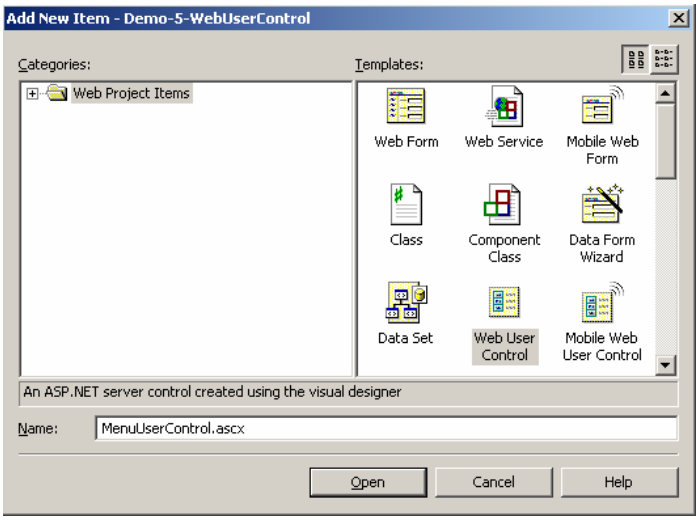
```
<%@ Page language="c#" Codebehind="MainForm.aspx.cs"
  AutoEventWireup="false"
  Inherits="Demo_4_WebUserControl.WebForm1" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
  <HEAD>
    <title>WebForm1</title>
    <meta name="CODE_LANGUAGE" Content="C#">
    <meta name="vs_defaultClientScript" content="JavaScript">
    <meta name="vs_targetSchema"
content="http://schemas.microsoft.com/intellisense/ie5">
  </HEAD>
  <body MS_POSITIONING="GridLayout">
    <form id="Form1" method="post" runat="server">
      <asp:Label id="LabelMain" style="Z-INDEX: 101; LEFT:
136px; POSITION: absolute; TOP: 16px" runat="server">Main Page
</asp:Label>
    </form>
  </body>
</HTML>
```

Както виждате на нея има само един етикет отбелязващ името на страницата (Main, Contacts, About).

Ще се заемем с направата на потребителската контрола:



Въвеждаме име за контролата:



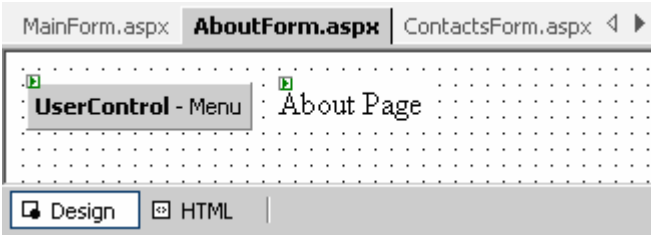
Получаваме файл с разширение **ascx**, в който има първоначално следният код:

```
<%@ Control Language="c#" AutoEventWireup="false"
  Codebehind="MenuWebUserControl.ascx.cs"
  Inherits="Demo_5_WebUserControl.MenuWebUserControl"
  TargetSchema="http://schemas.microsoft.com/intellisense/ie5"%>
```

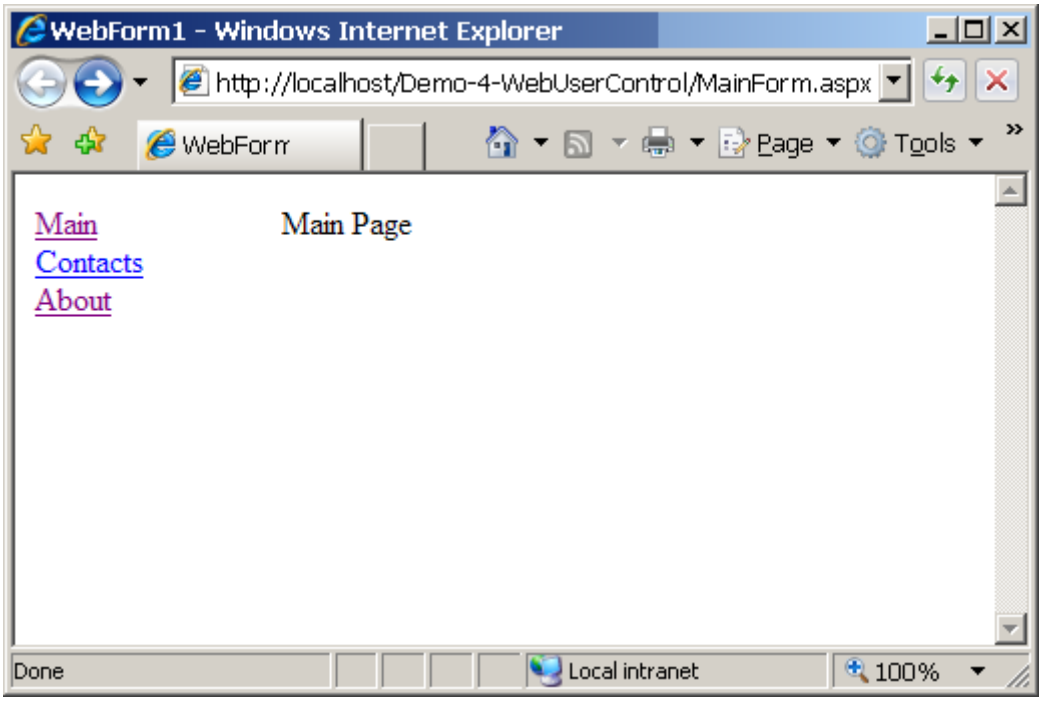
За да се появят бутони в менюто, добавяме следния код за три уеб контроли за бутони:

```
<p>
  <asp:HyperLink id="LinkMain" runat="server"
    NavigateUrl="MainForm.aspx">Main</asp:HyperLink>
  <br/>
  <asp:HyperLink id="LinkContacts" runat="server"
    NavigateUrl="ContactsForm.aspx">Contacts</asp:HyperLink>
  <br/>
  <asp:HyperLink id="LinkAbout" runat="server"
    NavigateUrl="AboutForm.aspx">About</asp:HyperLink>
</p>
```

Всеки бутон води до една от трите уеб форми, които създадохме. Следващата стъпка е да добавим новата контрола към всяка от трите уеб форми. Ето как изглеждат новополучените уеб форми:



Сега ни остава само да пуснем приложението и да проверим какво сме направили:



Всичко работи както трябва – менюто ни пренасочва към отделните страници.

Забележка: Контролите могат да се зареждат динамично с **LoadControl()** метода. Не е задължително да ги декларираме в **.aspx** страницата.

Проследяване и дебъгване на уеб приложния

За диагностициране на проблеми в уеб приложенията се използват две основни техники – проследяване (tracing) и дебъгване (debugging).

Информация по време на изпълнение

Докато уеб приложението работи, можете да събирате информация като използвате класовете **Trace** и **Debug**. Възможно е да извършвате следните действия по време на работа на приложението:

- да изписвате стойности на променливи;
- да разберете дали определени изисквания са изпълнени. Например методът **Trace.WriteIf(…)** изписва съобщение само когато е изпълнено дадено условие;
- да проследявате пътя на изпълнение на приложението. Можете да следвате програмната логика на дадена уеб форма, докато приложението се изпълнява, за да проверите дали всичко се извършва както очаквате.

Проследяване

Класовете **Trace** и **Debug** от пространството от имена **System.Diagnostics** са стандартния механизъм в .NET Framework за изписване (показване) на информация по време на изпълнение (runtime).

С **Trace** информацията се показва на самата уеб страница или се запазва в паметта. За да се следи състоянието на уеб приложението в традиционните ASP страници можеше да се използват методите **Response.Write** или изписване на debug информация в **Label** контроли на уеб формата. Предимството на **Trace** пред тези подходи е, че проследяването може да се контролира централизирано чрез настройките в конфигурационния файл **Web.config**. Така след като свършите с дебъгването на приложението си, можете лесно да изключите показването на информацията.

Методите на класа **Debug**, ще се изпълнят само ако приложението е компилирано в дебъг режим и е стартирано в дебъгер. Когато създавате release версия, извикванията няма да се изпълнят. С класа **Debug** можете да изписвате съобщения в Output прозореца на дебъгера на Visual Studio .NET. Използването на класа **Debug** не намалява надеждността на приложението, защото кодът не се променя - в release режим тези оператори просто не се изпълняват.

Проследяване на ниво страница и приложение

Проследяването може да ви помогне да диагностирате проблеми и да анализирате производителността. Можете да пишете директно в страницата или да запазвате trace информацията в база от данни.

При проследяване на ниво страница (page-level tracing), съобщенията се добавят в края на уеб страницата, за която е пуснато проследяването. При проследяване на ниво приложение (application-level tracing) съобщенията се добавят към всяка страница в приложението.

Използването на проследяване, пуснато само за отделна страница, позволява бързо да се види информацията от проследяването, докато се разглежда съдържанието на страницата. Когато стане ненужно, то може директно да бъде изключено, без да премахвате всички **`Trace.Write(…)`** оператори от кода.

Проследяването на ниво приложение (application-level tracing) се контролира от **`Web.config`** файла и дава повече гъвкавост. Например може съобщенията от проследяването да се пазят в паметта, и по-късно да се показват чрез използването на специалната страница **`trace.axd`**.

Категории на проследяване

Има няколко категории от информация, които се показват в Trace:

- **Request Details** - информация за заявката: идентификатор на сесията (ID), време на заявката, вид на заявката и статус на заявката;
- **Trace Information** - изход (Output) от стандартни и потребителски дефинирани trace оператори. Колоната "From First(s)" указва времето в секунди, откакто първото съобщение в тази секция е било показано. Колоната "From Last(s)" указва времето, изминало от показването на предишния ред. За яснота: за всеки два последователни записа (реда) имаме: From First(s) - From Last(s) на втория е равно на From First(s) на първия;
- **Control Tree** - списък на всички елементи, които са на страницата, с големината на всеки от тях;
- **Cookies Collection** - списък на всички използвани бисквитки (cookies);
- **Headers Collection** - списък на всички записи в HTTP хедъра;
- **Form Collection** - списък на контролите и техните стойности във формата (<form runat="server">...);
- **Server Variables** - списък на всички сървърни променливи: името на сървъра, текущо изпълняваната **`.aspx`** страница и т.н.

Обектът Trace

Освен класа **System.Diagnostics.Trace**, съществува и едноименно свойство на страницата **Trace**, което е от тип **TraceContext**. С негова помощ в секцията "Trace Information" освен показването на стандартна (предефинирана) информация от проследяването, можете да изписвате и произволни съобщения в определени от вас категории. Използват се методите **Trace.Write(…)** и **Trace.Warn(…)**, които работят по подобен начин, с единствената разлика, че **Trace.Warn(…)** изписва съобщенията в червено.

Динамичен контрол върху проследяването

Със свойството **Trace.IsEnabled** проследяването може динамично да се включва/изключва. Свойството е с по-голям приоритет от настройките за проследяване на ниво приложение.

Настройки на проследяването

Дори когато бъде пуснато проследяване на ниво приложение, настройките за проследяването на ниво страница се запазват. Например, ако се изключи проследяване за някоя страница, а проследяването за цялото приложение е пуснато, за страницата няма да се появи проследяваща информация. Следната таблица показва резултатите от различните комбинации проследяване на ниво приложение и на ниво страница:

На ниво страница	На ниво приложение	Резултат за конкретната страница
Trace=True	без значение	има проследяване (trace)
Trace=False	без значение	няма проследяване (trace)
не е указано	Trace=True	има проследяване (trace)
не е указано	Trace=False	няма проследяване (trace)

Атрибутът pageOutput

За указване къде да се показват съобщенията от проследяването можем да се използваме атрибута **pageOutput** на елемента **trace** във файла **Web.config**. Ако е **true**, съобщенията се показват на самата страница след края на съдържанието ѝ (добавят се отдолу). Ако е **false**, съобщенията се записват в паметта. Ето един пример за изключване на съобщенията от страницата (запазват се в паметта):

```
<configuration>
  <system.web>
    <trace enabled="true" pageOutput="false" />
  </system.web>
</asp:DropDownList>
```

Страницата trace.axd

Ако информацията от проследяването не се показва на страницата, тя се запазва в паметта. Може да бъде видяна, като се използва специална страница, която е включена по подразбиране във всяко уеб приложение. Адресът на страницата е: http://сървър/проект/trace.axd.

Поради причини свързани със сигурността, тази страница понякога е добра да бъде спряна. Това може да стане на ниво уеб сървър чрез конфигурационния файл machine.config. Той се намира в системната папка **C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\CONFIG**, като някои от директориите може да са с различни имена:

```
<httpHandlers>
  <add verb="*" path="trace.axd"
   type="System.Web.Handlers.TraceHandler">
</httpHandlers>
```

В горния пример, за да бъде спряна страницата, трябва атрибутът **path** да има за стойност празен текст (**path=""**).

Проследяване в потребителски компонент

Ако един компонент се вика от уеб форма, в него могат да се използват методите за проследяване (като **Trace.Write(…)** и **Trace.Warn(…)**). Това позволява да се генерират съобщения за проследяване (trace messages) за уеб формата и за компонента.

Когато се позволи проследяване в компонент, съобщенията се изписват в резултатите на всяка страница, която ползва компонента дори ако проследяването за тази страница е спряно.

Отдалечено дебъгване

Под отдалечено дебъгване (remote debugging) се разбира дебъгване на приложения на отдалечен сървър. Можете да дебъгвате от една работна станция ASP.NET приложения, изпълнявани на множество сървъри.

За отдалеченото дебъгване се изискват:

- Visual Studio .NET или неговите компоненти за отдалечено ползване, инсталирани на сървъра.
- Visual Studio .NET, инсталирано на работната станция.
- Административни права за сървъра.
- Акаунтът, използван за сървъра, да е в групата Debugger Users.

Стъпки за отдалечено дебъгване:

1. Стартира се Visual Studio .NET на клиентската машина.
2. File → Open → Project From Web.

3. В Open Project From Web диалоговата кутийка се пише адреса (URL) на сървъра.
4. В Open Project диалоговата кутийка се избира проектът на отдалечения сървър.
5. След като се отвори проектът, може да се използват breakpoints все едно приложението е локално.

Оптимизация, конфигурация и разгръщане на ASP.NET приложения

До момента разгледахме основните концепции и техники за разработка на ASP.NET уеб приложения. Сега, нека обърнем внимание на средствата за оптимизиране на уеб приложения чрез кеширане и на процеса на разгръщане на уеб приложение в средата, където трябва да работи (deployment), както и свързаните с това настройки на конфигурационни файлове.

Оптимизиране чрез кеширане

При изграждането на големи уеб приложения, които ще бъдат използвани едновременно от много потребители в рамките на минути или секунди, ние ще повтаряме едни и същи операции за всяка индивидуална заявка към нашето приложение. За да избегнем този повтарящ се процес, може да използваме кеширане. Кеширането е процес на запазване на често достъпвани данни (или такива, чието извличане отнема много ресурси) в паметта (или друго хранилище). Така те могат лесно и бързо да бъдат извлечени при повторно поискване.

Кеширане в ASP.NET

Кеширането е една от най-често използваните техники за оптимизация на ASP.NET приложение. В ASP.NET има два вида кеширане. Първият е кеширане на цялата **aspx** страница (генерирания HTML код) или части от нея. Вторият е кеширане на специфична за приложението информация, която ще бъде повторно достъпна за разработчика.

Кеширане на страница или отделни фрагменти от нея

Кеширането на ASP.NET страница се изразява в запазване на HTML кода, който тя е генерирала за определен период от време. При повторно извикване на същата страница, преди този период да е изтекъл, към клиентския браузър се изпраща вече генерирания HTML. Този процес значително подобрява бързодействието на приложението, като дори задаване на период от няколко секунди може да даде видим резултат.

За да укажем, че искаме дадена страница да се кешира, трябва да използваме директивата **@OutputCache**. Ето и пример, който указва, че дадената страница (или контрола) трябва да се кешира за 30 секунди:

```
<%@ OutputCache Duration="30" VaryByParam="None" %>
```

Същият резултат може да постигнем и в кода, който стои зад страницата. Ето пример как можем да направим това:

```
Response.Cache.SetExpires(DateTime.Now.AddSeconds(30));
Response.Cache.SetCacheability(HttpCacheability.Server);
```

Няма да се впускаме в подробности за разликата между двата начина, само ще спомена, че чрез методите на **HttpCachePolicy** (инстанция на този тип се връща от свойството **Cache** на **Response**) имаме достъп на ниско ниво до различните опции за кеширане. Докато чрез директивата **OutputCache** ни се предоставя едно добро ниво на абстракция, като ясно декларираме какво точно да се кешира.

Нека да разгледаме по-важните атрибути на директивата **@OutputCache**:

Атрибут	Описание
Duration	*Време за кеширане* Указва времето в секунди, за което дадената страница (потребителска контрола) ще се кешира. Атрибутът е задължителен.
VaryByParam	*Кеширане на версии по параметър* Чрез този атрибут може да кешираме няколко различни версии на страницата. Той ни позволява да зададем списък от параметри, разделени с точка и запетая, спрямо които да се кешират различните версии, понеже съдържанието на страницата (рендираният HTML) може да е различно спрямо даден параметър от query string, Атрибутът е задължителен. Негови стойности може да са ***** и **None**.
VaryByControl	*Кеширане на версии по ID на контрола* Атрибутът е подобен на предходния с изключение, че като стойност се задават ID на потребителските контроли, които искаме да кешираме.
Shared	*Кеширане между отделни страници* Този атрибут се указва само в потребителски контроли. Неговото предназначение е да укаже дали кешираната контрола може да се използва между отделните страници на приложението. Използва се при статични потребителски контроли, например лого или банер.

Кеширане на данни

Досега разгледаният метод за кеширане беше на ниво страници и генерирани от тях HTML. Сега ще разгледаме другия вид за кеширане в ASP.NET, а именно кеширането на информация (обекти), която да бъде лесно достъпна при повторно поискване. Това е възможно благодарение на класа **`System.Web.Caching.Cache`**, който служи като контейнер (речникова колекция) за обекти, които ще бъдат използвани повторно. Нека да разгледаме някои от предимствата и недостатъците на **`Cache`** класа, след което ще се спрем на различните начини за добавяне на обекти в кеша и тяхното унищожаване (invalidation).

Предимства:

- Осигурява бърз достъп до обекти, чието създаване е бавно, скъпо или отнемащо много ресурси (извличане от база данни, уеб услуга, криптирано устройство и др.).
- Поддържа автоматично заключване на обекта, който се използва. Това позволява безопасна конкурентна работа над този обект.
- Предлага разнообразни опции за унищожаване на обектите в него (дори и за тяхното обратно създаване чрез callback функции).
- Автоматично започва да унищожава кешираните обекти, когато ресурсите на сървъра намалеят.

Недостатъци/забележки:

- Може да се използва в рамките на едно приложение, т.е. всяко едно приложение има свой кеш, който е единствен и не може да бъде споделян с останалите приложения.
- Горното ограничение ефективно води до загуба на скалируемост. Обектите са тясно свързани с приложението, работещо на конкретния сървър и не могат да бъдат споделяни между сървъри в уеб ферми.
- **`Cache`** контейнерът е активен (жив), докато приложението работи. При рестартиране на приложението **`Cache`** обектът се създава отново.
- **`Cache`** контейнерът не може да съхранява данни за конкретен потребител. За тази цел се използва сесията (**`HttpSessionState`**) или речниковата колекция **`HttpContext.Items`**, ако искаме да запазим информация само за текущата заявка.

Кеширане на данни – примери

Както вече споменахме, добавянето на обекти в кеша може да стане по няколко начина с различни политики за унищожението на добавяния обект.

Стандартният начин е да се обърнем към кеша като речникова колекция. Ето един пример:

```
DataSet dsUsers = GetAllUsers();
Cache["UsersDataSet"] = dsUsers;
```

Извличането на вече добавен обект също е стандартно:

```
DataSet dsUsers = (DataSet) Cache["UsersDataSet"];
```

Ако обектът междувременно е бил унищожен, се връща **null**.

Политики за унищожаване на обектите при кеширане

Да разгледаме по-подробно метода **Insert(…)** на **Cache**. Този метод има няколко дефиниции с различен брой параметри, които може да използваме, за да задаваме различни политики относно това кога да се унищожи добавяният обект. Ето примери за използването на всяка една от тях:

- **Унищожаване на добавяния обект след определен период от време**. Следният код добавя обект, който ще бъде унищожен след 5 минути:

```
Cache.Insert("myKey", myValue, null, DateTime.Now.AddMinutes(5),
  Cache.NoSlidingExpiration);
```

- **Унищожаване на добавяния обект след определен период от време от последното му използване**. Следният код добавя обект, който ще бъде унищожен 20 секунди, след като е бил използван. Ако в следващите 20 секунди отново извлечем този обект от кеша, отчитането на секундите започва отначало:

```
Cache.Insert("myKey", myValue, null, Cache.NoAbsoluteExpiration,
  TimeSpan.FromSeconds(20));
```

- **Унищожаване на добавяния обект при дадена зависимост (промяна на файл или унищожаването на друг обект от кеша)**. В последните два примера третия параметър, който подаваме на метода, е **CacheDependency**. Чрез конструкторите на този клас можем да укажем изтриване на добавяния обект при промяна на даден файл (съвкупност от файлове) или при унищожаването на друг обект (съвкупност от обекти) от кеша.

  Ето пример, който илюстрира как добавяният обект ще се унищожи, когато файлът **myConfig.xml** бъде променен:

```
Cache.Insert("myKey", myValue,
  new CacheDependency(Server.MapPath("myConfig.xml")));
```

- **Задаване на приоритет на добавяния обект**. Друга възможност, която ни се предоставя, е да зададем приоритет на добавяния обект. Когато сървърът започне да освобождава ресурси, сравнява

приоритетите на всички обекти и унищожава тези с най-нисък приоритет. Възможните приоритети са стойностите на изброимия тип **CacheItemPriority – Low, BelowNormal, Normal (Default), AboveNormal, High, NotRemovable**. В следващия пример обектът, който добавяме, ще е един от последните унищожени:

```
Cache.Insert("myKey", myValue, null, Cache.NoAbsoluteExpiration,
  Cache.NoSlidingExpiration, CacheItemPriority.High, null);
```

- **Извикване на callback функция, когато даден обект бива унищожен**. Кеш класът ни предоставя и възможност за извикване на наша callback функция. За целта трябва да създадем инстанция на делегат от тип **CacheItemRemovedCallback**. Ето пример:

```
public void RemovedCallback(string aKey, object aValue,
  CacheItemRemovedReason aCallbackReason )
{
   switch ( aCallbackReason )
   {
      case CacheItemRemovedReason.Expired :
        //do work when item is expired
        break;
      case CacheItemRemovedReason.DependencyChanged :
        //do work when item's dependency changed
        break;
      default:
        break;
   }
}

private void CacheItem( string aKey, object aItem )
{
   CacheItemRemovedCallback onRemove =
     new CacheItemRemovedCallback(RemovedCallback);

   Cache.Insert( aKey, aItem, null, Cache.NoAbsoluteExpiration,
     Cache.NoSlidingExpiration, CacheItemPriority.Default,
     onRemove );
}
```

Възможността за извикване на callback функции може да се използва и за да се постави обекта отново в кеша.

Конфигуриране на ASP.NET приложение

Конфигурацията на едно ASP.NET приложение се извършва на основата на съвкупност от няколко XML базирани конфигурационни файла. Изнасянето на конфигурационните настройки в отделен файл (а не в кода) дава изключително лесна процедура за разгръщане на приложението (XCOPY

Deployment). Това позволява и промяна на някои от настройките, без да се налага прекомпилация.

Файлът Machine.config

Всеки ASP.NET уеб сървър има свой глобален конфигурационен файл – **Machine.config**. Той се намира в: **systemroot\Microsoft.NET\Framework\<versionNumber>\CONFIG\Machine.config**, където **systemroot** стандартно е **C:\WINDOWS**, а **versionNumber** е **v1.1.4322** за .NET Framework 1.1 В този файл се съдържат глобалните настройки (настройки по подразбиране). Те се прилагат върху всяко едно уеб приложение. Няма да се спираме подробно на тях , само ще споменем, че в **Machine.config** се съдържат и глобалните настройките за **machineKey**, Той служи за криптиране и хеширане на ViewState и бисквитката за сесията. В случай, че имаме приложение, което работи в web-farming среда (на няколко сървъра), трябва да сме подсигурим, че стойностите на **machineKey** на всеки един от сървърите са еднакви.

Неправилна промяна на файла Machine.config може да окаже влияние на всички уеб приложения, които работят на сървъра.

Файлът Web.config

Освен глобалния конфигурационен файл за сървъра всяко едно отделно приложение има свой собствен конфигурационен файл – **Web.config**. Той вече ни е познат, защото когато създадем нов уеб проект, това е един от файловете, който автоматично е добавен в него. Ето как изглежда той в редактора на Visual Studio:

Web.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

  <system.web>

    <compilation defaultLanguage="c#" debug="true" />

    <customErrors mode="RemoteOnly" />

    <authentication mode="Windows" />

    <authorization>
        <allow users="*" /> <!-- Allow all users -->
    </authorization>

    <trace
        enabled="false"
        requestLimit="10"
        pageOutput="false"
        traceMode="SortByTime"
        localOnly="true"
    />
```

Във файла **Web.config** се указват специфичните настройки за приложението, като някои от тях може да препокриват тези от файла **Machine.config**. Всички настройки са разположени йерархично в различни секции или категории. За да разберем какво точно може да конфигурираме, нека да разгледаме по-значимите от тях.

Категории в Web.config

Настройките, които можем да зададем във файла **Web.config**, свързани с работата на уеб приложението, се намират в секцията **system.web**. Ето нейните по-важните подсекции:

Секция	Описание
authentication	Избор на метод на автентикация и неговите свойства. Подробно ще се спрем на автентикация в частта "Сигурност".
authorization	Предоставя възможност за декларативно прилагане на сигурността, базирана на роли (role-based security) и оторизацията на потребителите.
browserCaps	Възможност за задаване на филтри, спрямо които браузъра, направил заявката, може да се разпознае и асоциира.
compilation	Настройки, указващи по какъв начин да се компилира приложението, когато дойде първата заявка към него.
customErrors	Възможност за конфигуриране как ASP.NET да се справя с възникналите грешки и изключения.
globalization	Настройки на глобализацията на приложението, в това число културата на приложението, кодирането на файловете, заявките и отговорите, направени от и към сървъра.
httpHandlers	Предоставя възможност за асоцииране на класове, които да обработят заявки към дадени ресурси.
httpModules	Предоставя възможност за добавяне на допълнителни модули, които да предоставят дадена функционалност. Сесията, автентикацията и оторизацията са реализирани като такива модули.
identity	Възможност за имперсонация на текущия потребител, с който се асоциират заявките към сървъра.
pages	Предоставя възможност да се променят настройките по подразбиране за всички страници в приложението.
processModel	Богат набор от настройки за изпълнението на прило-

	жението от IIS, включително дали да се използва уеб ферма.
sessionState	Разнообразни настройки за сесията – дали да се използват бисквитки, дали сесията да бъде съхранена в SQL сървър и др.
trace	Настройки за проследяването на приложението – дали да се проследява, да се показва ли дневникът (log) на страницата и др.

Забележка: Съдържанието на **Web.config** е чувствително към малки и главни букви.

Разширяемата структура на файла Web.config

Както вече разгледахме, файлът **Web.config** ни предоставя богата възможност за конфигуриране на отделните части от приложението. Но всяка разгледана до сега настройка беше стандартно предоставена от ASP.NET. Как обаче да съхраним наша специфична информация за приложението в конфигурационния файл? За тази цел може да използваме специалната секция в **Web.config** файла – **appSettings**. В нея може да задаваме двойки ключ-стойност. Те са достъпни програмно по време на изпълнение на приложението. Ето примерен конфигурационен файл:

```
<configuration>
  <system.web>
    ...
  </system.web>
  <appSettings>
    <add
      key="ConnectionString"
      value="server=demoserver;database=pubs;uid=sa;pwd=" />
    <add
      key="MailServer"
      value="DemoHost" />
  </appSettings>
</configuration>
```

Извличането на тези стойности става по следния начин:

```
string connectionString = System.Configuration.
  ConfigurationSettings.AppSettings["ConnectionString"];
SqlConnection conn = new SqlConnection(connectionString);
. . .
SmtpMail.SmtpServer = System.Configuration.
ConfigurationSettings.AppSettings["MailServer"];
```

ASP.NET ни дава възможност да изграждаме наши собствени конфигурационни секции в файла **Web.config**. Чрез тях можем да структурираме

конфигурационните настройки на приложението и да групираме в отделни блокове логически свързаните.

Йерархия на конфигурационните файлове

Всяка директория в уеб приложение може да съдържа свой собствен конфигурационен файл (**Web.config**), в който може да се предефинират настройките за тази директория и всички нейни поддиректории. По този начин се получава йерархия на конфигурационните настройки и файлове. Най-отгоре стои глобалният конфигурационен файл за сървъра - **Machine.config**. Неговите настройки се наследяват от главния конфигурационен файл за приложението (файла **Web.config**, разположен в главната директория). Те се прилагат върху всички поддиректории.

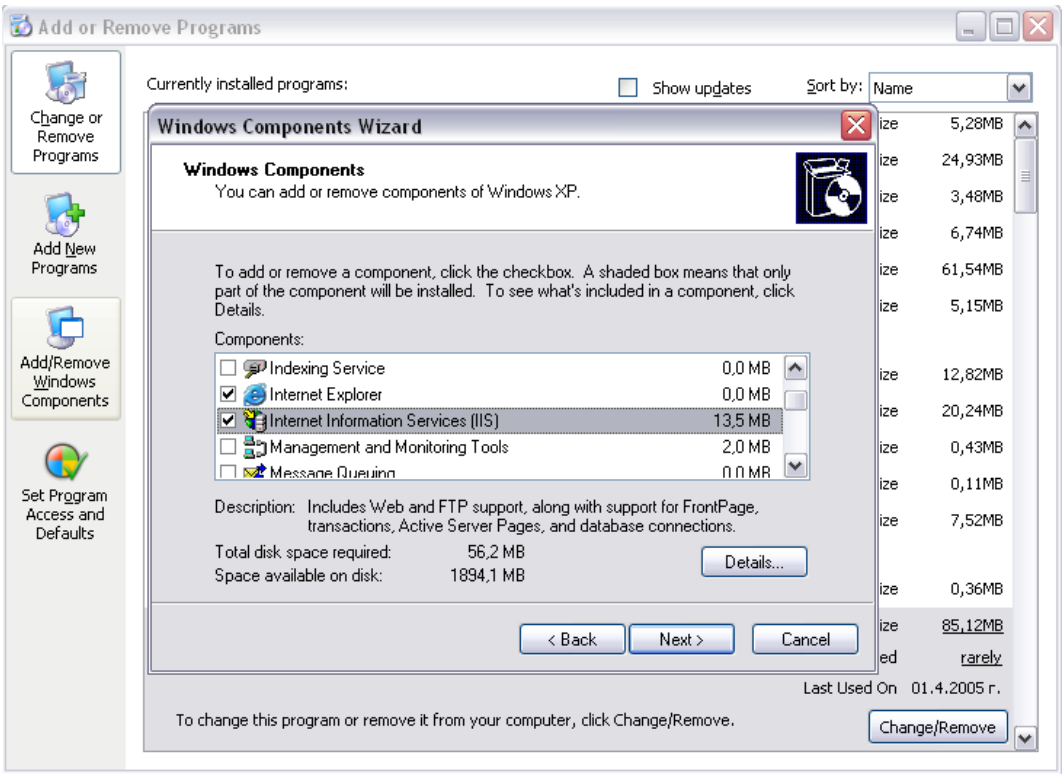
Разгръщане на приложението

След като вече разгледахме какви са възможностите за конфигуриране на приложението, сега ще спрем вниманието си върху неговото разгръщане (deployment) и последващата го поддръжка и обновяване. Но малко преди това ще проследим стъпките за инсталиране и конфигуриране на уеб сървъра.

Инсталиране и конфигуриране на уеб сървъра

Уеб сървърът (IIS – Internet Information Services) не е инсталиран стандартно в Windows 2000 или Windows XP (нещата не стоят така при Windows Server 2000 и 2003). За да го инсталираме, трябва да направим следното.

1. Отваряме **Control Panel** и избираме **Add or Remove Programs**.
2. От появилия се прозорец избираме етикета **Add/Remove Windows Components**.
3. От новопоявилия се списък с компоненти на операционната система избираме и инсталираме **Internet Information Services (IIS)**.



Ако успешно сме извършили гореописаната операция, ще трябва да рестартираме Windows. След рестартиране, от **Control Panel -> Administrative Tools -> Internet Information Services** можем да отворим интерфейса за конфигуриране на сървъра.

В случай, че сме инсталирали Visual Studio .NET преди IIS (или изобщо нямаме Visual Studio), ще е необходимо да регистрираме ASP.NET работния процес. За целта трябва да въведем следния ред в командния интерпретатор на Visual Studio, намиращ се в неговото подменю в **Start** менюто:

```
aspnet_regiis -i
```

Това може да стане и като стартираме файла **aspnet_regiis.exe**, който се намира в **systemroot\Microsoft.NET\Framework\versionNumber** с параметър **-i**.

Необходими файлове

Както всички .NET приложения, така и ASP.NET уеб приложенията се разгръщат чрез просто копиране (XCOPY deployment). Необходимите файлове, които трябва да копираме във виртуалната директория на приложението, са:

- папката **bin**, която съдържа компилираните code-behind класове и всички асемблита, които сме реферирали в нашия проект.
- всички уеб форми (***.aspx**) и потребителски контроли (***.ascx**)
- конфигурационните файлове на приложението (**Web.config**) и файла за обработка на глобални събития (**Global.asax**).
- всякакви други допълнителни файлове, които използва приложението – картинки, лицензни файлове и др.
- ако приложението използва динамична компилация, ще са ни нужни и code-behind файловете (***.aspx.cs** и ***.ascx.cs**).

Всички останали файлове, които се намират в директорията на приложението, не са необходими (***.sln**, ***.csproj**, ***.resx**). Както вече споменахме, ако не използваме динамична компилация, code-behind файловете също няма да са ни необходими.

Обновяване на приложението

Обновяването на уеб приложението се извършва чрез копиране на всички променени страници и потребителски контроли, както и на асемблито, което съдържа компилираните code-behind класове. Ако има промени в конфигурационните файлове на приложението, те също трябва да бъдат обновени. Работният процес на ASP.NET следи за промени в bin директорията и конфигурационните файлове и ако настъпят такива, автоматично рестартира приложението. След рестартиране първият потребител, който поиска дадена страница, ще предизвика JIT компилация на приложението.

Сигурност в ASP.NET

Концепцията за сигурност е залегнала в основата на ASP.NET. Уеб приложенията, които изграждаме, по всяка вероятност ще се ползват от много на брой потребители и сигурно ще са достъпни през Интернет. Това изисква от ASP.NET да предложи добре развит механизъм за осигуряване на сигурност.

Сигурността в ASP.NET се основава на цялостната система за сигурност в .NET и в частност на модела, базиран на роли (Role-Based Security). ASP.NET предлага модели за автентикация (authentication) и оторизация (authorization), които заедно с предоставените услуги от уеб сървъра (IIS) изграждат цялостната инфраструктура за сигурността в ASP.NET. Въпреки че в темата за сигурност, ще разгледаме автентикацията и оторизацията, нека и сега се спрем на тези две дейности.

Автентикация и оторизация

Преди да разгледаме в детайли как се извършва автентикацията в ASP.NET и оторизацията при достъпа до защитени ресурси, нека обясним първо какво означават термините "автентикация" и "оторизация".

Автентикация

Автентикацията е процесът на разпознаване на даден потребител. Потребителят се представя като предоставя данни за себе си (напр. Потребителско име и парола). Тези данни се проверяват за валидност. Ако са валидни, потребителят се счита за автентикиран. В противен случай му се отказва достъп до система или поискания ресурс. В ASP.NET има три възможности за автентикация: windows, forms и passport автентикация. Ще се спрем по-подробно на всяка от тях след малко.

Оторизация

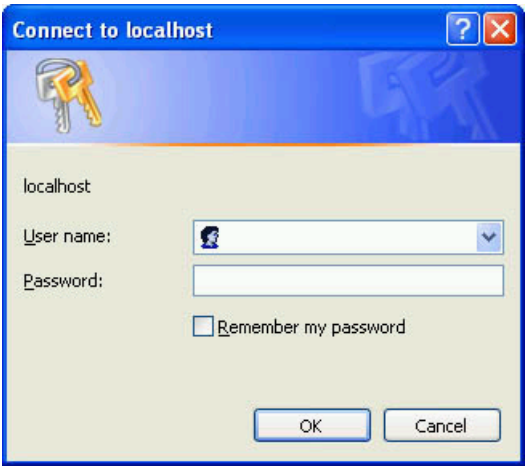
Оторизацията е процес на свързване на потребител с дадени права. За оторизиран се счита потребител, който има право да работи с поискания ресурс или да извърши конкретната операция. Във веригата на сигурността това е следващият процес след автентикацията – след като разберем кой е потребителят, ние трябва да знаем какви са неговите права. В ASP.NET за оторизация се използва моделът Role-Based Security, т.е. всеки потребител може да е в една или повече роли. Процесът на оторизация може да се извършва не само на ниво потребител, но и на ниво роля.

Видове автентикация в ASP.NET

Както вече споменахме, в ASP.NET има три вида автентикация: **windows**, **forms** и **passport** (всъщност са четири, но четвъртият е **none** – никаква). Ще разгледаме всеки един от тях, като се спрем на неговите предимства и недостатъци. Ще обсъдим в кои ситуации кой модел да използваме.

Windows автентикация

Windows автентикацията разчита на самата операционна система да предостави информация дали даденият потребител е този, за който се представя. За целта, ако дадена страница е достъпна само за автентикирани потребители, пред потребителя се появява диалогов прозорец. В него той трябва да въведе име и парола:



Така въведените данни се проверяват за валидност спрямо потребителите на сървъра или на домейна, в който той се намира. Ако са валидни, потребителят се счита за автентикиран.

Как ще се запази информацията, че даден потребител вече е автентикиран, зависи от настройките, които направим на уеб сървъра. Възможностите са следните: basic, digest и integrated оторизация. Нека да разгледаме всяка от тях накратко:

Basic автентикация

Това е най-простият метод за автентикация и най-непрепоръчителният, защото паролата се предава в чист вид в HTTP хедъра на всяка заявка. Предимствата на този метод са, че е официално приет стандарт и се поддържа от всички съвременни браузъри.

Digest автентикация

Подобна е на basic автентикацията с едно единствено предимство – името и паролата не се предават в чист вид. Въпреки това изисква самите пароли да са в чист вид (или криптирани) на сървъра, което означава, че достъпът до него трябва да е ограничен.

Integrated Windows автентикация

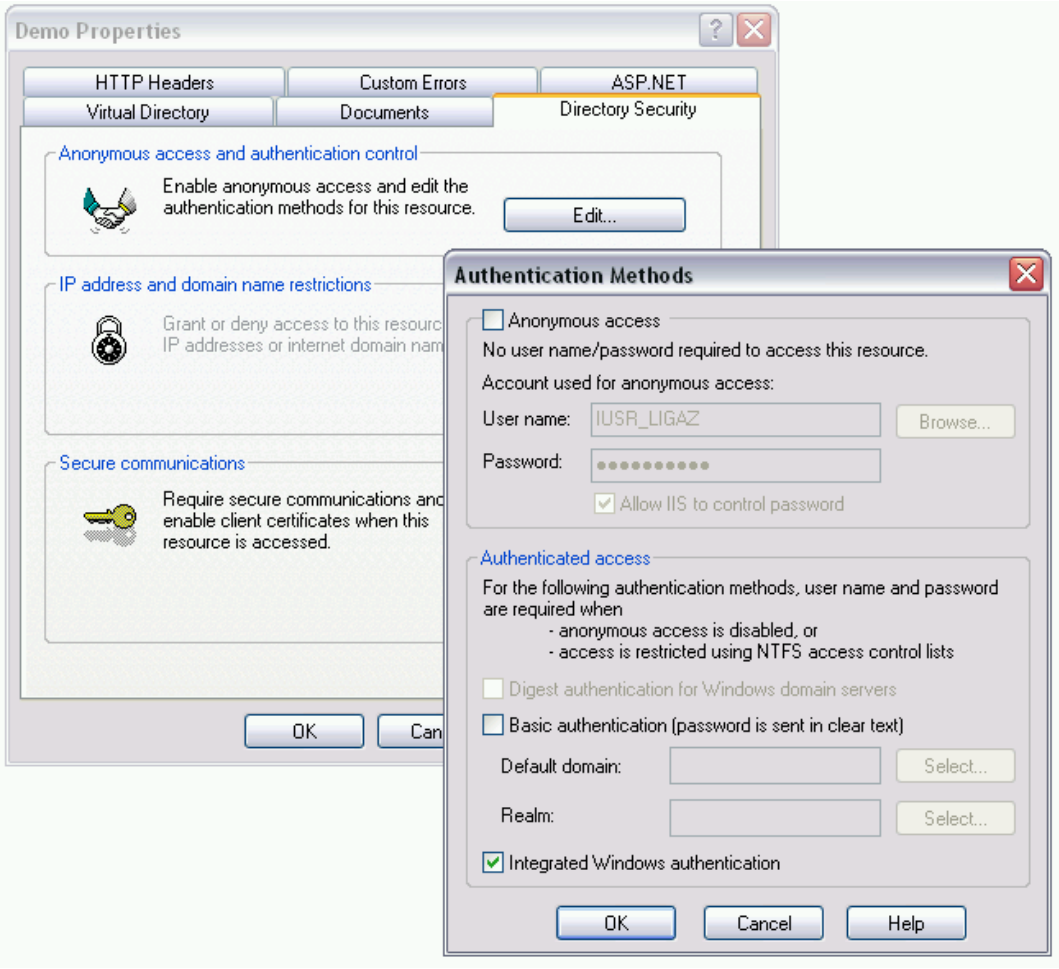
Това е най-сигурният метод за автентикация в Windows среда. При него не се предава никаква конфиденциална информация (няма диалогов прозорец за въвеждане на данни), а потребителят се автентикира като текущо влезлия (logged) потребител в операционната система, от която идва заявката. Естествено сигурността и удобството имат своята цена – тази възможност се поддържа само от Internet Explorer (уеб сървъра и браузъра осъществяват комуникацията по свой собствен начин). Използват се портове, различни от 80, за да се осъществи автентикацията, което може да е проблем, ако има защитна стена (firewall) в мрежата.

След като разгледахме всяка една от възможностите за Windows автентикация, нека да видим как да зададем коя да използваме.

За начало указваме да се използва Windows автентикация в конфигурационния файл на приложението **`Web.config`**:

```
<authentication mode="Windows" />
```

След това отваряме конфигурационната конзола на IIS и с десен бутон върху нашия проект избираме **Properties**. От появилия се прозорец избираме етикета **Directory Security** и натискаме бутона **Edit**, който се намира в първата секция: **Anonymous access and authentication control** (вж. фигурата). След това имаме възможност да изберем необходимия ни метод – в случая сме избрали Integrated Windows автентикацията.



Windows автентикацията е най-добре да използваме, ако разработваме приложение, което ще се използва в рамките на една компания (в нейния Интранет), където потребителите са част от потребителския домейн и са фиксиран брой. Този вид автентикация е неприложим, ако приложението ще се използва в Интернет.

Forms автентикация

Това е може би най-често използваният метод за автентикация в ASP.NET. В него самото приложение се грижи за автентикирането на потребителите. След малко ще разгледаме подробен пример как да използваме този вид автентикация, а сега нека разгледаме принципа, на който тя се базира.

Forms автентикация – принцип на действие

При поискване на ресурс (страница), който е разрешен само за автентикирани потребители, клиентският браузър се пренасочва към предварително указана страница, на която ще се извърши автентикацията. При успешна автентикация към клиента се изпраща бисквитка, която указва,

че потребителят е вече автентикиран. При всяка следваща заявка бисквитката се прихваща и използва от ASP.NET за разпознаване на автентикираните потребители.

Forms автентикацията е най-масово използваният метод за автентикация, защото е много удобен за реализиране на конкретна логика за управление на потребителите. Този метод е и най-удобен, ако разработваме приложения, които ще се ползват в Интернет, където броят на потребителите е силно динамичен. Единственото неудобство е, че разчита на бисквитки, но и затова е помислено, като има възможност за сесия без бисквитки – cookieless session.

Автентикация и оторизация чрез Forms authentication и Role-based security – пример

В следващия пример ще разгледаме как може да използваме Forms автентикацията в реална ситуация. Като начало ще зададем използването на Forms автентикация във файла **Web.config**:

```
<authentication mode="Forms" >
  <forms loginUrl="Login.aspx" />
</authentication>
```

Атрибутът **loginUrl** се използва, за да укажем на коя страница ще се автентикира потребителят. При поискване на страница, изискваща автентикация, потребителят ще бъде пренасочен към **Login.aspx**, където ще може да се автентикира. Другата настройка, която трябва да направим в конфигурационния файл, е да укажем кои ресурси ще изискват автентикация. Ето фрагмент от конфигурационен файл, който дефинира всички уеб форми под директорията **Admin** да изискват автентикирани потребители:

```
<configuration>
  <system.web>
    ...
  </system.web>
  <location path="Admin">
    <system.web>
      <authorization>
        <deny users="?" />
      </authorization>
    </system.web>
  </location>
</configuration>
```

Същото може да се постигне, като поставим **Web.config** в директорията **Admin** със следното съдържание:

```
<configuration>
```

```
  <system.web>
    <authorization>
      <deny users="?" />
    </authorization>
  </system.web>
</configuration>
```

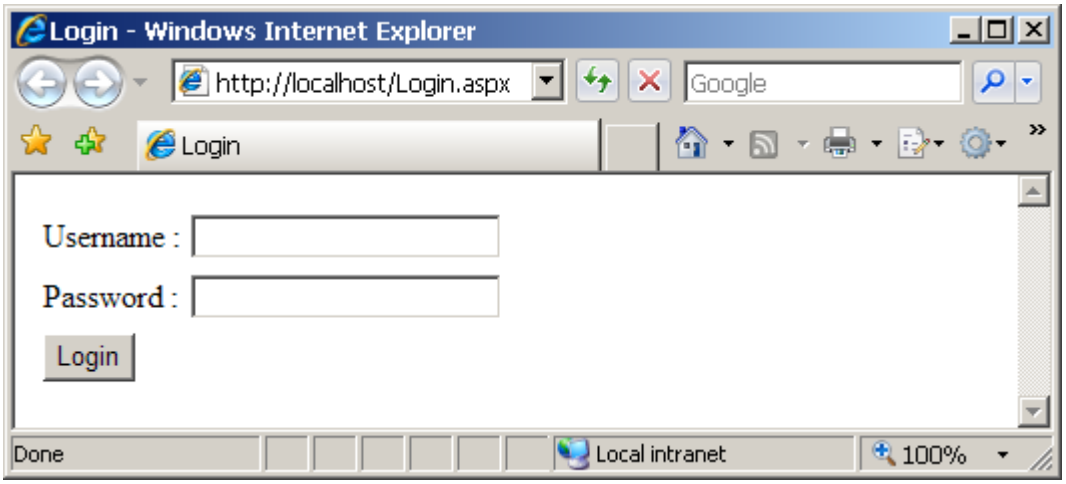
Нека се спрем малко по-подробно на секцията **authorization** и нейните дъщерни елементи. Елементът **deny** отказва достъпа до този ресурс на съответните потребители или роли, като за потребители се използва атрибутът **users**, а за роли – **roles** (ако разрешените са повече от една, те са разделени със запетая). Аналогично има елемент **allow**, който разрешава достъпа. Като стойности на тези атрибути могат да се използват и знаците ***** (всички потребители) и **?** (потребителите, които не са автентикирани). Когато ASP.NET проверява дали потребител има достъп до даден ресурс, правилата се прилагат отгоре надолу. Ако се стигне до правило, което му разрешава или отказва достъп, то се изпълнява, а стоящите под него се игнорират. Ако няма такова, се счита, че потребителят има достъп до поискания ресурс.

Сега ще разгледаме как ще изглежда нашата форма за влизане в системата. Ето съществената част от **Login.aspx** файла:

```
<form id="LoginForm" method="post" runat="server">
  <table border="0" cellSpacing="2" cellPadding="2">
    <tr>
      <td>Username :</td>
      <td>
        <asp:TextBox id="TextBoxUsername" runat="server" />
      </td>
    </tr>
    <tr>
      <td>Password :</td>
      <td>
        <asp:TextBox id="TextBoxPassword" runat="server"
          TextMode="Password" />
      </td>
    </tr>
    <tr>
      <td colspan="2">
        <asp:Button id="ButtonLogin"
          runat="server" Text="Login" />
      </td>
    </tr>
    <tr>
      <td colspan="2">
        <asp:Label id="LabelErrorMessage" runat="server" />
      </td>
    </tr>
```

```
   </table>
</form>
```

Ето и как ще изглежда формата в клиентския браузър:



Сега ще разгледаме кода, който извършва автентикацията. Методът, който обработва събитието **Click** на бутона **ButtonLogin**, е в code-behind файла на формата **Login.aspx**:

```
private void ButtonLogin_Click(object sender, EventArgs e)
{
   if (TextBoxUsername.Text == TextBoxPassword.Text)
   {
      FormsAuthentication.RedirectFromLoginPage(
       TextBoxUsername.Text, false );
   }
   else
   {
      LabelErrorMessage.Text = "Invalid login!";
   }
}
```

На първия ред извършваме наивна валидация на потребителското име и парола, като ги сравняваме дали са равни. В реална ситуация ще ни се наложи да се обърнем към базата от данни или да извикаме уеб услуга, за да установим дали данните са валидни. В случай, че са валидни, трябва да извикаме статичния метод **RedirectFromLoginPage(…)** на класа **FormsAuthentication**. Той приема два параметъра: потребителското име, което ще се запише в бисквитката за автентикация и флаг, дали тази бисквитка да остане за определено време при клиента (продължителността се конфигурира в **Web.config**). Вторият параметър служи да се избегне операцията по автентикация, ако затворим браузъра. Методът пренасочва потребителя към първоначално поискания от него ресурс, който е

изисквал автентикация. Ако искаме да го пренасочим на друго място, трябва да използваме друг статичен метода **SetAuthCookie(…)**, който само изпраща бисквитката за автентикация. Друг полезен метод на класа **FormsAuthentication** е **HashPasswordForStoringInConfigFile(…)**. Той служи за хеширане на потребителските пароли. Ако потребителят не е въвел правилно своите данни, изписваме съобщение за грешка.

Не съобщавайте на потребителите дали са сбъркали само името или само паролата. Това може да ги насочи към потребителските имена на съществуващи потребители и да доведе до пробиви в сигурността на приложението.

След като проследихме как става автентикацията, нека да разгледаме как се извършва оторизацията чрез сигурност, базирана на роли. Единственият код, който трябва да напишем за целта, е в **Global.asax.cs** файла:

```
protected void Application_AuthenticateRequest(Object sender,
  EventArgs e)
{
   if (HttpContext.Current.User != null)
   {
      if (HttpContext.Current.User.Identity.IsAuthenticated)
      {
         FormsIdentity identity =
           HttpContext.Current.User.Identity as FormsIdentity;
         if (identity != null)
         {
            if (identity.Name == "Stefan" )
            {
               HttpContext.Current.User = new GenericPrincipal(
                 identity, new string[]{ "Web Developer" } );
            }
         }
      }
   }
}
```

Методът **Application_AuthenticateRequest** се извиква, когато даден потребител бъде автентикиран. Ето какво правим в този случай. Проверяваме дали наистина е автентикиран и ако е така, проверяваме дали се използва Forms автентикация. Ако такава е налична, може да използваме свойството **Name** на обекта **identity**, което ни връща вече съхраненото име за потребителя в бисквитката за автентикация. След това реализираме логиката за задаване ролите на потребителя, който се е автентикирал. В случая на потребителя "Stefan" се задава роля "Web Developer", което ще му позволи достъп до всички ресурси, които са разрешени както за него, така и за неговата роля.

Passport автентикация

Този метод се базира на услугата MS Passport, която Microsoft предлага на своите клиенти. Тази услуга всъщност представлява голямо единно хранилище на информация за регистрирали се потребители. Информацията за тях е достъпна през уеб услуги. Идеята на тази услуга е, че потребителят влиза в системата само веднъж и след това може да влиза директно и в други сайтове, използващи същата автентикация. Излизането може да стане както от текущия сайт, така и от всички сайтове, в които е влязъл потребителят. Предимствата на този подход са, че се предоставя единен механизъм за работа с потребители (единна база от данни), както и че има високо ниво на сигурност. Недостатъците са, че услугата не е безплатна, а и работата на приложението става зависимо от трета страна (в случая Microsoft).

Сигурност на ниво сървър (IIS Security)

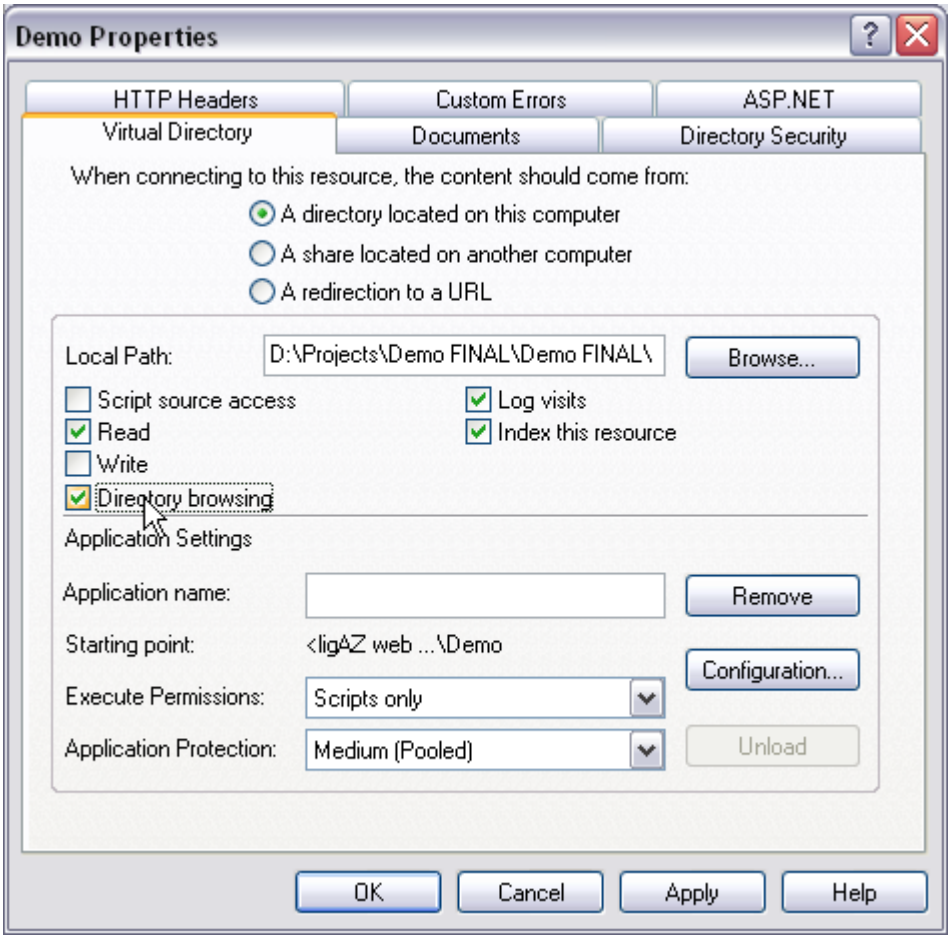
За финал ще разгледаме какво ни предоставя IIS сървъра за осигуряване на сигурност на приложението. Основното предназначение на един уеб сървър е да обслужва заявките, направени от клиентските браузъри към ресурси, които се намират на сървъра. Поисканият ресурс може да не съществува или клиентът да няма право да го види.

Разглеждане файловете на сървъра

Стандартно IIS разрешава достъпа само до определени ресурси (**`*.aspx`**, **`*.html`**, **`*.jpg`** и др.), останалите файлове не се обслужват (напр. **`Web.config`**, **`*.cs`** и др.).

Ако е необходимо отдалечено разглеждане на файловете на приложението, може да го разрешим, като маркираме настройката **Directory Browsing**, намираща се в менюто **Properties**, щраквайки с десен бутон върху уеб приложението в потребителския интерфейс на IIS.

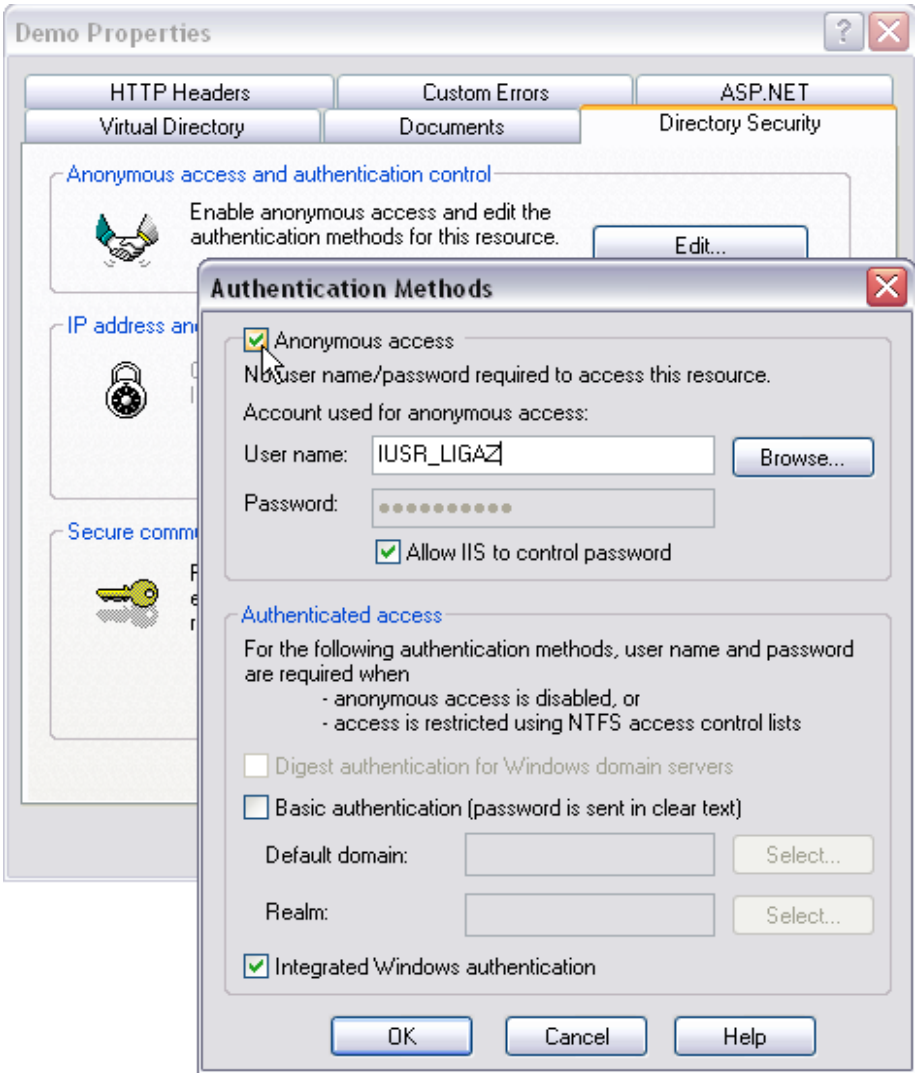
На фигурата по-долу е показан диалогът за настройка на "Directory Browsing" опцията.



Анонимен достъп

Поискването на ресурс от файловата система на уеб сървъра трябва да се идентифицира с валиден потребител на системата. Всяка заявка, направена от клиентски браузър към ресурс, за който е разрешен анонимен достъп, се идентифицира като анонимна (стига да не е направена чрез Internet Explorer, чийто потребител е в мрежата на сървъра) и се асоциира със служебния потребител **`IUSR_machinename`**, където machinename е името на сървъра. Този потребител се добавя в системата при инсталацията на сървъра.

Ако искаме да разрешим анонимен достъп до файловете и/или да променим потребителя, с който се асоциира анонимния достъп, отново трябва да щракнем с десен бутон върху приложението и да изберем **Properties**.



Този път трябва да изберем **Directory Security** и щракаме върху бутона **Edit**, който се намира в първата секция: **Anonymous access and authentication control.**

	**Не давайте по-големи права от необходимите за достъп на акаунта IUSR_machinename.**

Криптиране на връзката чрез SSL

Уеб сървърът (IIS) предлага и възможност за криптиране на връзката, като за целта се използва най-разпространеният стандарт Secure Sockets Layer (SSL). Стандартно браузърът и сървърът комуникират като си пращат информацията в прав текст. Използвайки SSL сертификат, двете страни по сигурен начин обменят ключ, който ще се използва за криптиране на комуникацията между тях. Дори и недоброжелател да прихване

предаваната информация, той няма да е в състояние да ги декриптира (поне в разумни срокове и с нормални технически средства). За да се използва SSL, на сървъра трябва да се инсталират необходимите сертификати. Те могат да бъдат издадени единствено от определените органи за това (Certification Authorities). Стандартно SSL комуникацията протича на порт 443 и може да се познае по това, че адресът на сайта започва с **`https://`**.

Упражнения

1. Създайте уеб проект. Разгледайте генерираните файлове и обяснете за какво служи всеки един от тях. Покажете code behind файловете. Отпечатайте "Hello world" в **`aspx`** файл и в code behind файл. Покажете файловете, автоматично генерирани в папката **`Assembly.GetExecutingPath()`**.
2. Създайте HTML форма, която предава съдържанието на **`textarea`** поле към сървъра и сървърът го отпечатва в ново поле. Не използвайте code-behind файл.
3. Създайте ASP.NET уеб форма, която предава съдържанието на **`textarea`** поле към сървъра, който го отпечатва в ново поле.
4. Използвайте **`src`** атрибута на **`@Page`** директивата, за да направите страница, която няма нужда от компилация.
5. Създайте уеб форма, която по параметри зададени в GET заявката да определя широчината на текстова кутия, адреса на хипервръзка и височината на картинка. Формата да се направи в два варианта – с помощта на HTML и на Web сървърни контроли.
6. Създайте уеб форма, която да има две текстови полета и един бутон. При натискане на бутона да се извърши проверка на клиентската страна дали двете текстови полета имат еднаква стойност и само тогава формата да се подаде на сървъра.
7. Създайте уеб форма с текстово поле и бутон. При натискането на бутона отпечатайте въведения текст в контрола от тип **`Label`** и в друг Web server контрола от тип **`TextBox`** в режим **`MultiLine`**. Въведете в текстовото поле некоректни символи и отстранете HTML escaping проблема, където той се появява. Обяснете работата на контролите.
8. Прихванете събитията за всички етапи от живота на страниците с помощта на методи и реализирайте проследяване за тях.
9. Създайте потребителска контрола, който да визуализира меню. Контролата трябва да има свойства за инициализация на менюто – двумерен масив съдържащ името и страницата на съответния елемент. Имплементирайте свойство, което да определя цвета и шрифта на менюто. Преценете има ли нужда от ViewState поддръжка.

10. Създайте HTML страница, която да отпечатва типа на браузъра, IP-то и порта, който клиента използва, за да отвори страницата.

11. Създайте уеб страница, която да запазва съдържанието на текстово поле в **Session** обекта и да го отпечатва в поле от тип етикет.

12. Създайте две страници, които да си предават информация въведена от потребителя чрез бисквитка. Бисквитката трябва да е валидна 5 мин.

13. Създайте страница, която да показва таблица, в която на всеки ред има разположени **Label** контроли и един бутон. При натискане на бутона, **Label** контролите на текущия ред да се скрият и да се покажат **TextBox** контроли с текущото съдържание на **Label** контролите. При повторно натискане на бутона, да се върне първоначалното състояние. Да не се използват по-усложнение контроли като **DataGrid**, **DataList** и подобни.

14. Създайте форма за регистрация на потребители с данни за име, имейл, парола, повтаряне на паролата, телефон и опция за съгласие с общите условия на сайта. Всички полета са задължителни и съобщенията за грешки да се извеждат в обща контрола. Полетата за имейл и телефон да се валидират с регулярен израз, а двете полета за парола да се проверяват дали са с еднаква стойност.

15. Създайте уеб форма, която съдържа **DataGrid** контрола. Реализирайте свързване с таблици от базата от данни Northwind и реализирайте избор, редактиране и триене на редове. сортиране и страниране на резултатите.

16. Визуализирайте данните от таблица с помощта на Repeater контрола.

17. Създайте уеб сайт с "login" страница, страница за административен достъп и страница за публичен достъп. Реализирайте и "logout" функционалност. Използвайте Forms authentication и роли на потребителите.

18. Създайте уеб страница, която да има три бутона и едно поле – етикет. С единият бутон да се инициализира **Cache** обекта със стойност, която "остарява" след 10 секунди. С вторият бутон – стойност, която да "остарява" 10 секунди след настоящия момент. С третият бутон да се извежда стойността на този елемент от кеша и да се показва в етикета.

19. Създайте потребителска контрола, който да използва изходящо кеширане със зависимост по елемент от **Cache** обекта.

20. Покажете идентичността на процеса, който изпълнява ASP.NET проекта, при модел на работа на IIS 5.1 и IIS 6.0 с помощта на следните методи:

    - **Page.User.Identity.Name;**

    - **System.Security.Principal.WindowsIdentity.GetCurrent().Name;**

- **`System.Threading.Thread.CurrentPrincipal.Identity.Name`**;

21. Създайте уеб страница, която да създаде празен файл в **`Program Files`** папката. Конфигурирайте правилно правата на папката, така че да бъде разрешено писането на IIS процеса.

Използвана литература

1. Михаил Стойнов, ASP.NET уеб базирани приложения – http://www.nakov.com/dotnet/lectures/Lecture-15-ASP.NET-and-Web-Applications-v1.01.zip
2. MSDN Documentation - http://msdn.microsoft.com/
3. World Wide Web Consortium (W3C) - The HTML Coded Character Set – http://www.w3.org/MarkUp/html-spec/html-spec_13.html както и по-пълен списък http://www.natural-innovations.com/wa/doc-charset.html
4. Jeff Prosise, Programming Microsoft .NET, Microsoft Press, 2002, ISBN 0735613761
5. Andrew Duthie , Microsoft ASP.NET Programming with Microsoft Visual C# .NET Version 2003 Step by Step, Microsoft Press, 2003, ISBN 0735619352

www.devbg.org

Българска асоциация на разработчиците на софтуер (БАРС) е нестопанска организация, която подпомага професионалното развитие на българските софтуерни специалисти чрез образователни и други инициативи.

БАРС работи за насърчаване обмяната на опит между разработчиците и за усъвършенстване на техните знания и умения в областта на проектирането и разработката на софтуер.

Асоциацията организира специализирани конференции, семинари и курсове за обучение по разработка на софтуер и софтуерни технологии.

БАРС организира създаването на Национална академия по разработка на софтуер – учебен център за професионална подготовка на софтуерни специалисти.

Глава 17. Многонишково програмиране и синхронизация

Автори

Александър Русев

Иван Митев

Необходими знания

- Базови познания за .NET Framework и CLR
- Базови познания за общата система от типове в .NET (Common Type System)
- Базови познания за езика C#
- Базови познания по операционни системи
- Атрибути

Съдържание

- **Многозадачност**
  - Проблемът – защо многозадачност?
  - Ползите от многозадачността
  - Решението – процеси и нишки
  - Какво предлагат нишките и кога са удобни?
  - Видове многозадачност
  - Имплементации на многозадачност
  - Домейни на приложението
- **Нишки**
  - Как работят нишките?
  - По-важни членове на класа **`Thread`**
  - Приоритет на нишките
  - Състояния и живот на нишките
  - Thread Local Storage
  - Thread-Relative Static Fields

  - Повреждане на данни и други неудобства
- **Синхронизация**
  - Най-доброто решение
  - Стратегии за синхронизация
  - Синхронизирани пасажи код
  - Синхронизирани контексти
  - **`MethodImplAttribute`**
  - Неуправлявана синхронизация – **`WaitHandle`**
  - Класически синхронизационни проблеми
- **Пул от нишки - `ThreadPool`**
- **Интерфейсът `ISynchronizeInvoke`**
- **Таймери**
- **Асинхронни извиквания**
  - Асинхронни извиквания на методи и приложения
  - Асинхронно извикване чрез делегат
  - Модел за асинхронни извиквания
  - Интерфейсът **`IAsyncResult`**
  - Приключване на асинхронен метод

В тази тема ...

В настоящата тема ще разгледаме многозадачността в съвременните операционни системи и средствата за паралелно изпълнение на програмен код, които ни предоставя .NET Framework. Ще обърнем внимание на нишките (threads), техните състояния и управлението на техния жизнен цикъл – стартиране, приспиване, събуждане, прекратяване и др.

Ще разгледаме средствата за синхронизация на нишки при достъп до общи данни, както и начините за изчакване на зает ресурс и известяване при освобождаване на ресурс. Ще се спрем на синхронизационните обекти в .NET Framework, както и на неуправляваните синхронизационни обекти на операционната система.

Ще изясним концепцията за работа с вградения в .NET Framework пул от нишки (thread pool), начините за асинхронно изпълнение на задачи, средствата за контрол над тяхното поведение и препоръчваните практики за работа с тях.

Многозадачност

В тази първа точка от темата ще обясним какво е многозадачността и какъв смисъл има от нея. Казано накратко, многозадачността е възможността на процесора да разпределя времето си върху повече от една задача.

Проблемът

Често на едно приложение се налага да извършва времеотнемащи операции. Докато те се изпълняват, потребителят трябва да бъде известяван за статуса на работа. Той трябва да е наясно дали приложението продължава да извършва обработки или е блокирало.

В други случаи едно приложение трябва да изчаква освобождаването на споделен ресурс, за да може да продължи работата си. Този и горният сценарии демонстрират необходимостта от механизъм, който да позволява поддръжка на паралелно изпълнение на няколко операции.

Ползите от многозадачността

В случаите на многопроцесорни системи многозадачността води до повишена производителност. Когато изпълнението на приложението е разделено на части, които могат да бъдат изпълнени независимо една от друга, то те могат да се разпределят между процесорите и да приключат за по-малко време.

В еднопроцесорните системи, многозадачността е не по-малко важна, защото позволява на приложението да взаимодейства по-добре с потребителя, като постоянно го известява за състоянието си и е способно да отговаря на действията му във всеки момент.

Многозадачността е много полезна и когато една система се използва от много потребители едновременно. Разпределянето на процесорното време между потребителите, чрез помощта на нишките, създава за всеки един от тях илюзията, че работи сам с приложението. Същевременно не се изразходват излишни системни ресурси за поддържане на цял процес за всеки потребител.

Защо е нужна многозадачност – пример

Нека разгледаме за пример приложение, което при натискане на бутон изпълнява времеотнемаща операция. През това време, потребителският интерфейс не отговаря, тъй като приложението е заето с изчисления.

При стартиране на програмата, виждаме следното:

Графичният интерфейс на приложението се състои само от едно текстово поле, в което потребителят да въвежда произволен текст и бутон, в обработката на който стои следната времеотнемаща операция:

```
private void buttonStartJob_Click(object sender,
   System.EventArgs e)
{
   // Start the job in the current thread
   new TimeTakingJob().Job();
}
```

Класът **`TimeTakingJob`** има следната реализация:

```
class TimeTakingJob
{
   public void Job()
   {
      long sum = 0;
      for (int i=0; i<100000; i++)
      {
         for (int j=0; j<100000; j++)
         {
            if (i==j)
            {
               sum++;
            }
         }
      }
   }
}
```

Забелязваме, че функцията изпълнява два вложени цикъла, които водят до едно продължително изчисление.

Как работи примерът?

Когато потребителят натисне бутона, това тежко изчисление започва да се изпълнява в главната нишка на приложението. Ще дефинираме какво

точно е нишка в следващата точка, засега можем да считаме нишката за част (единица) от приложението.

Резултатът от натискането на бутона е замръзване на потребителския интерфейс. Приложението е заето с продължителни изчисления и не може да обработи никакви действия на потребителя докато не приключи със сметките.

Алтернативата

За да се избегне този проблем, кодът, който се изпълнява при натискането на бутона, трябва да изглежда подобно на следния:

```
private void buttonStartJob_Click(object sender,
   System.EventArgs e)
{
   // Start the job in a seperate thread
   Thread t = new Thread(
      new ThreadStart(new TimeTakingJob().Job));
   t.Start();
}
```

Тогава изчислението се пуска в отделна нишка и потребителският интерфейс реагира коректно. Работа на процесора е да разпредели времето си между нишката за изчислението и нишката за интерфейса.

Решението – процеси и нишки

Процесът е съвкупността от памет, стек и код на приложението. Операционната система работи с процеси, които потребителите възприемат като приложения - това са две имена за едно и също понятие. Както видяхме в предишния пример, един процес може да изисква паралелно изпълнение на повече от една задача. Затова процесите са съставени от една или повече нишки, които се изпълняват едновременно от гледна точка на потребителя (всъщност тази илюзия се постига, като процесорът често и бързо превключва между тях). Нишката е основната единица, за която се заделя процесорно време.

Процеси и нишки

Ще се спрем на приликите и разликите между процесите и нишките.

Както процесите, така и нишките, имат собствен стек и имат определен приоритет. Процесите са независими един от друг по отношение на памет и данни. За разлика от тях, всички нишки в един процес споделят обща памет – паметта на процеса, към който принадлежат.

Докато процесите съдържат изпълнимия код, нишките го изпълняват – процесите са пасивни, а нишките – активни.

Какво предлагат нишките?

Използването на няколко нишки създава впечатление за извършване на много задачи едновременно. Причината е, че процесорът се предоставя на всяка нишка за някакъв определен интервал от време (квант). Разпределянето на времето се осъществява на базата на различни стратегии.

След изтичането на този квант, се получава прекъсване и процесорът се предоставя на следващата чакаща нишка. Прекъсването е механизъм, позволяващ нормалната последователност от процесорни инструкции да бъде променена. Този тип прекъсвания са известни като софтуерни прекъсвания – те са предварително планирани и синхронни с работата на процесора. Освен тях, процесорът може да получава и хардуерни прекъсвания, които са асинхронни, т. е. могат да постъпят в произволен момент. Те също водят до промяна в изпълняваната последователност от инструкции.

Именно механизмът на прекъсванията през достатъчно малък интервал от време създава впечатлението за едновременно изпълнение на повече операции. Когато например потребителят въвежда данни в текстообработваща програма, други данни могат да се печатат на принтер. Би било неудобно за потребителя ако не може да върши друга работа, докато принтерът работи.

Кога са удобни нишките?

Удобно е да се ползват нишки при обслужване на много потребители едновременно, напр. при приложение от тип уеб сървър. Когато потребител се свърже, се пуска нова нишка, чрез която да работи. Аналогично е и свързването с база от данни, всяка връзка към нея се обслужва от отделна нишка.

При мрежова комуникация (напр. през сокети), комуникацията може да бъде изолирана в отделна нишка и докато приложението чака отговор от другата страна, да извършва друга полезна работа.

Всяка нишка има приоритет. Нишките с по-висок приоритет заемат процесора по-често. Така можем да определяме приоритети на отделните задачи в едно приложение.

Изпълняването на дълги изчисления (като в примера), винаги трябва да става на заден план, за да може потребителският интерфейс да реагира на потребителски заявки.

Многозадачност – видове

Съществуват два вида многозадачност – кооперативна и изпреварваща.

При **кооперативната многозадачност** (cooperative multitasking), всяка нишка сама решава колко процесорно време й е необходимо. Веднъж заела процесора, тя го освобождава само ако приключи работата си или трябва да чака за някакъв ресурс – напр. дадено събитие или вход от

потребителя. Това обаче може да доведе до безкрайно отлагане (starvation) на останалите нишки и те да чакат неопределено дълго. В чистия й вид, кооперативната многозадачност има много ограничени приложения.

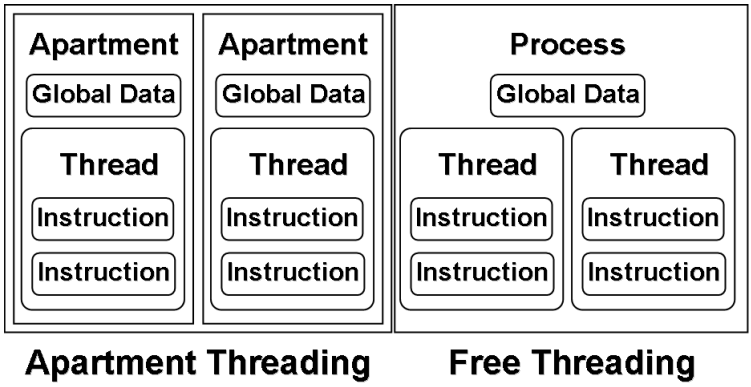
При **изпреварващата многозадачност** (preemptive multitasking), за всяка нишка предварително се заделя процесорно време. Системен софтуер, наречен планировчик (task scheduler), е отговорен за това разпределение на времето. В края на всеки такъв предварително зададен интервал от време, нишката се снема от процесора, без значение дали е приключила работата си.

В съвременните операционни системи (Windows 2000, Windows XP), се използва изпреварваща многозадачност. Тя е по-безопасна, тъй като при нея процесорът не може да бъде зает от една нишка за неопределено време и няма риск от безкрайно отлагане за останалите.

Някои системи използват комбиниран вариант – нишките с висок приоритет заемат процесора до приключването си (кооперативно), а останалите – на интервали (изпреварващо).

Имплементации на многозадачност

Многозадачността може да бъде имплементирана по два начина – самостоятелна многозадачност (Apartment Threading) и свободна многозадачност (Free Threading).



При самостоятелната многозадачност, всеки процес получава копие на данните, нужни за неговото изпълнение. Всяка нишка се стартира в неин собствен процес, така че няма споделени данни между нишките в един процес. Всяка работа, която искаме да извършим в нишка, се извършва в отделен процес. Тази многозадачност е извънпроцесна (out-of-process).

При свободната многозадачност данните в процеса са споделени между нишките и процесорът може да смени нишката, като в същото време не сменя данните, с които се работи.

Свободната многозадачност (Free Threading) е по-ефективното решение и затова се използва по-често в практиката. В .NET Framework нишките използват именно Free Threading модела.

Самостоятелна многозадачност

При модела на самостоятелната многозадачност (STA) всяка нишка "живее" в отделен апартамент в рамките на процеса. Процесът може да има произволен брой апартаменти и те да споделят данни помежду си чрез посредник (proxy). Приложението решава кога и за колко дълго трябва да се изпълнява нишката във всеки апартамент. Всички заявки се сериализират чрез Windows опашка със съобщения, така че по всяко време се достъпва само един апартамент и следователно само една нишка се изпълнява. STA е моделът, който познават повечето Visual Basic разработчици, защото преди появата на VB.NET само той е бил достъпен за VB приложенията.

Свободна многозадачност

При свободната многозадачност (MTA) данните в процеса са споделени между нишките и процесорът може да смени нишката, като в същото време не сменя данните, с които се работи. Този подход се използва често, защото позволява повишена ефективност. В .NET Framework се поддържа именно Free Threading модела, но за взаимодействие с COM има предвидени начина за работа със STA.

Домейни на приложението (Application Domains)

Когато се стартира едно .NET приложение, операционната система създава неуправляван процес. Приложението обаче не може да се изпълнява директно в неуправлявания процес. Затова се въвежда допълнително ниво на абстракция между приложението и процеса, наречено домейн на приложението. Домейнът е логическо понятие, за разлика от процеса, който е физически. Един неуправляван процес съдържа един или повече управлявани домейни на приложението.

Домейни на приложението – предимства

- По принцип процес не може да ползва данни на друг процес. Това ограничение може да бъде заобиколено с употребата на посредник (proxy), но това обикновено става за сметка на усложняване на кода. Използвайки домейни на приложението, можем да стартираме повече от едно приложение в един и същ процес. Така споделянето на данни между приложенията бива значително улеснено
- Домейните на приложението допълнително се разделят на контексти. Контекстът е също логическо понятие. Обектите, опериращи в един контекст, са контекстно свързани обекти. За контекстно свързаните обекти, .NET Framework предоставя допълнителен механизъм за синхронизация, който ще бъде разгледан в точката за синхронизация.

- Домейните на приложението поддържат проверка на типа на данните, които съдържат.

Също като процесите, домейните на приложението могат да съдържат една или повече нишки.

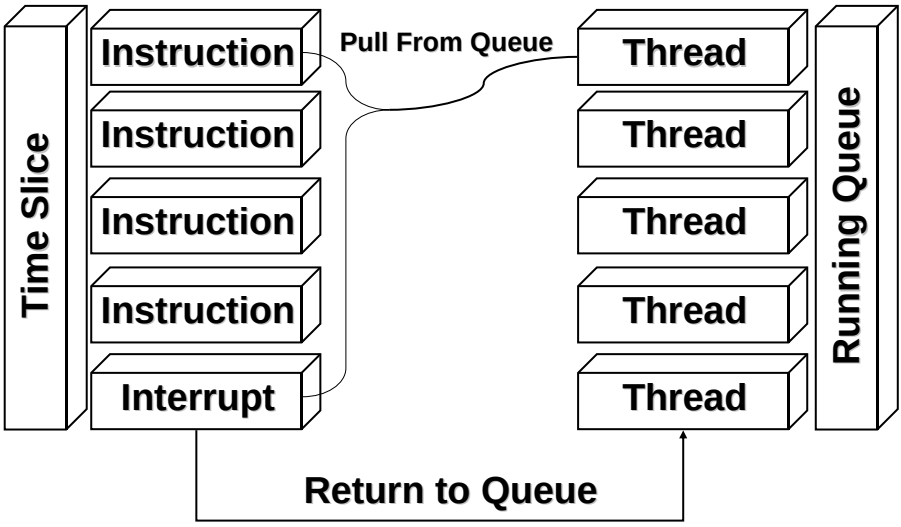
За достъп до домейн на приложението .NET Framework предоставя класа **`System.AppDomain`**.

Нишки

Нишките (threads) предоставят възможност на процесора да изпълнява няколко задачи едновременно, като паралелното изпълнение се симулира чрез постоянно превключване между задачите през много кратки интервали от време. Всяка нишка изпълнява някаква задача (програмен код) като от време на време заема процесора за много кратко време, след което го освобождава за изпълнение на друга нишка.

Как работят нишките?

Нека разгледаме принципната схема на работа на планировчика на задачите (task scheduler), който разпределя процесорното време между всички активни нишки.



От схемата се вижда, че в даден момент се поддържат известен брой текущо изпълнявани нишки (в дясната колона). Тъй като процесорът е един, те са подредени в опашка и всяка изчаква своя ред. Когато една нишка получи достъп до процесора, на нея се предоставя квант от време (time slice). Той започва с поредната за изпълнение процесорна инструкция и завършва с инструкция за прекъсване, което е знак за процесора да запомни регистрите на нишката, която е изпълнявал (т. е. да запази докъде е стигнало изпълнението на нишката). Междувременно, нишката

се връща в опашката, откъдето се избира следващата за изпълнение. Тя започва от там, до където е стигнала при последното си заемане на процесора и процесът се повтаря циклично.

Изпълнение на няколко нишки – пример

Ще дадем следния пример за демонстрация:

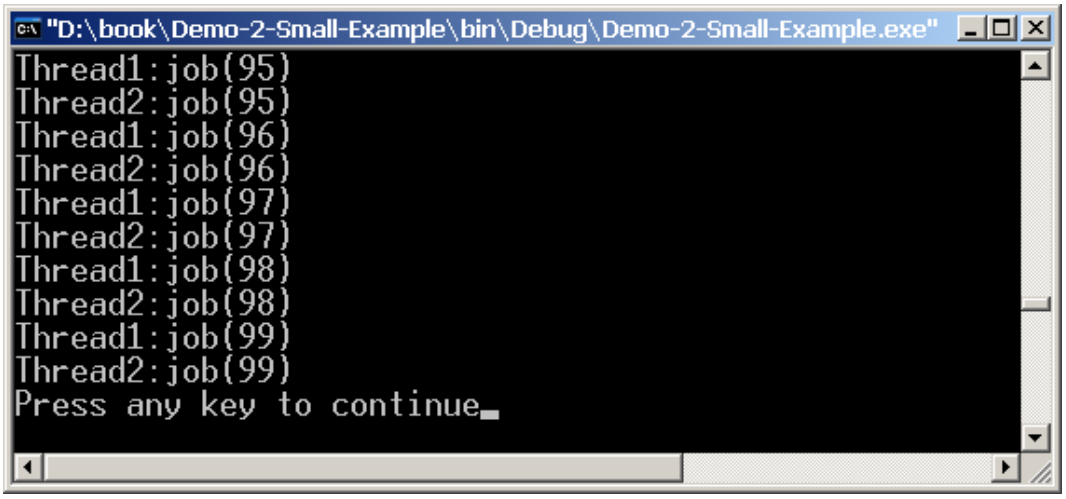
SmallExample.cs

```
using System;
using System.Threading;

namespace SmallExample
{
   class ThreadClass
   {
      public void DoTask1()
      {
         for( int i=0; i<100; i++ )
         {
            Console.WriteLine("Thread1:job({0})",i);
            Thread.Sleep(1);
         }
      }

      public void DoTask2()
      {
         for( int i=0; i<100; i++ )
         {
            Console.WriteLine("Thread2:job({0})",i);
            Thread.Sleep(1);
         }
      }
   }

   class MainThread
   {
      static void Main(string[] args)
      {
         ThreadClass threadClass = new ThreadClass();
         Thread thread1 = new Thread(
            new ThreadStart(threadClass.DoTask1));
         Thread thread2 = new Thread(
            new ThreadStart(threadClass.DoTask2));
         thread1.Start();
         thread2.Start();
      }
   }
}
```

Как работи примерът?

Главната нишка на приложението започва изпълнение от метода **`Main(…)`** на класа **`MainThread`**. Със създаването на два обекта от клас **`Thread`**, създаваме две нишки. При създаването на нишка, подаваме като параметър метода, от който тя да започне изпълнението си. В случая, това са методите **`DoTask1()`** и **`DoTask2()`** на класа **`ThreadClass`**. **`ThreadStart`** е делегат, който определя сигнатурата на метода - тяло на нишката, а именно – метод без параметри, който не връща стойност.

С извикването на метода **`Start()`** на двете нишки, всяка от тях започва да се изпълнява и върху конзолата започва да се изписва коя до къде е стигнала. При стартиране на примера се вижда, че често двете нишки приключват почти едновременно, тъй като изпълняват еквивалентен код.



Класът Thread

В .NET Framework за изпълнение на нишки се използва класът **`System.Threading.Thread`**. Този клас предоставя функционалност за стартиране и управление на нишки. Нека разгледаме неговите по-важни членове:

Thread (ThreadStart)

Създава инстанция. Подава се делегат с метод, който да се изпълни при стартиране. Създаването на нишка вече бе демонстрирано.

Sleep(...)

"Приспива" текущата нишка за указания брой милисекунди. Методът е статичен и блокира текущо изпълняваната нишка. След изтичането на зададения интервал, тя продължава работата си.

Suspend()

Ако нишката работи, я преустановява временно. Ако е преустановена, не се случва нищо. За разлика от **`Sleep()`**, чрез който нишка преустановява себе си за някакъв фиксиран интервал от време, **`Suspend()`** преустановява нишка за неопределено време и тя остава в това състояние до извикването на **`Resume()`**, който подновява изпълнението ѝ.

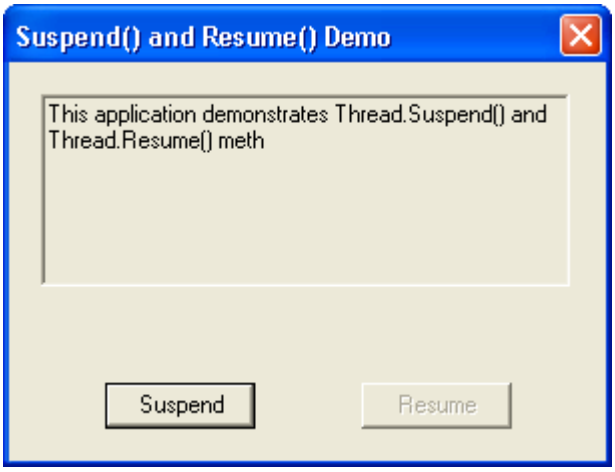
Resume()

Подновява нишка, която е била преустановена (suspended). Ако нишката работи, не прави нищо.

Некоректното използване на Suspend() и Resume() може да доведе до синхронизационни проблеми. Ако две нишки взаимно се чакат за Resume(), нито една няма да може да продължи и ще се стигне до "мъртва хватка" (deadlock).

Suspend() и Resume() – пример

Като пример за **`Suspend()`** и **`Resume()`**, ще дадем едно кратко Windows Forms приложение. При стартиране то печата даден текст буква по буква със забавяне между отделните букви. Визуално приложението изглежда по следния начин:



Нека разгледаме съществената част от сорс кода на примерното приложение:

```
delegate void CharParamDelegate(char aChar);
private const string MESSAGE="This application demonstrates " +
   "Thread.Suspend() and Thread.Resume() methods. ";

private Thread mThread;

private System.Windows.Forms.TextBox textBoxMessage;
```

```
private System.Windows.Forms.Button buttonResume;
private System.Windows.Forms.Button buttonSuspend;

private void MainForm_Load(object sender, System.EventArgs e)
{
   mThread = new Thread(new ThreadStart(this.PrintMessages));
   mThread.IsBackground = true;
   mThread.Start();
   SuspendThread();
}

private void SuspendThread()
{
   mThread.Suspend();
   buttonSuspend.Enabled = false;
   buttonResume.Enabled = true;
}

private void ResumeThread()
{
   mThread.Resume();
   buttonSuspend.Enabled = true;
   buttonResume.Enabled = false;
}

private void AppendTextToTextBox(char aChar)
{
   textBoxMessage.AppendText(aChar.ToString());
}

/// <summary>
/// PrintMessages() runs in a separate thread and slowly
/// prints messages in the MainForm's text box.
/// </summary>
private void PrintMessages()
{
   while (true)
   {
      foreach (char letter in MESSAGE.ToCharArray())
      {
         try
         {
            this.Invoke(new CharParamDelegate(
                 AppendTextToTextBox), new object[]{letter});
         }
         catch (Exception)
         {
            // Can not call Invoke() bacause the form is closed.
            return;
         }
```

```
        Thread.Sleep(50);
      }
   }
}

private void buttonSuspend_Click(object sender,
   System.EventArgs e)
{
   SuspendThread();
}

private void buttonResume_Click(object sender,
   System.EventArgs e)
{
   ResumeThread();
}
```

Как работи примерът?

При зареждане на формата, се пуска една нишка, която във вечен цикъл изписва даден текст в текстово поле символ по символ. Преди да я пуснем, установяваме в **true** свойството й **IsBackground**, с което я пускаме във фонов режим. Така тя ще спре автоматично при приключване на главната нишка, т. е. при затварянето на формата. Двата бутона викат съответно методите **Suspend()** и **Resume()** и определят в дадения момент кой бутон да бъде позволен.

Отпечатването на всеки отделен символ минава през метода **Form.Invoke(…)**. По този начин потребителският интерфейс на приложението се променя единствено от главната нишка на приложението.

Не променяйте графичния потребителски интерфейс от външна нишка. Последствията могат да бъдат непредсказуеми: забавяне, "зависване", повреда на данни и др.

IsAlive

Свойството **IsAlive** има стойност **true**, след като нишката се стартира. Нормалното приключване на нишката или прекратяването й поради външна намеса променят стойността на **IsAlive** на **false**. Повече информация за състоянията, през които една нишка преминава, дава **ThreadState**.

IsBackground

Свойство за четене и запис. Една нишка може да е на преден (foreground) или заден (background) план.

Когато всички нишки на преден план в един процес приключат, той приключва. CLR вика **Abort()** за всички нишки на заден план (известни още като нишки, работещи във фонов режим).

IsThreadPoolThread

Свойство за четене и запис. Има стойност **true**, ако нишката принадлежи на управлявания пул от нишки, иначе е **false**.

Name

Свойство за четене и запис на името. Всяка нишка в .NET Framework може да има име. Това свойство е полезно за идентифицирането на нишките при дебъгване и извеждане на диагностични съобщения.

Priority

Свойство за четене и запис на приоритета на нишката. Възможните стойности са **Lowest**, **BelowNormal**, **Normal** (по подразбиране), **AboveNormal** и **Highest**.

ThreadState

Свойство само за четене. Съдържа състоянието на нишката. Състоянията, в които една нишка може да попадне, ще бъдат подробно обяснени в следващата точка – засега можем да считаме, че състоянието на една нишка определя например дали текущо тя работи или изчаква.

Abort()

Хвърля **ThreadAbortException** в извиканата нишка, с което обикновено прекратява нишката. При определени условия, **Abort()** може и да не прекрати нишката. Това ще бъде обяснено в точка "Прекратяване".

Interrupt()

Ако нишката е в състояние **WaitSleepJoin**, хвърля **ThreadInterrupted-Exception**. Нишката може да прихване това изключение и да продължи изпълнението си. Ако тя не го прихване, CLR го прихваща и прекратява нишката.

Ако нишката не е в състояние **WaitSleepJoin**, извикването на **Interrupt()** не прави нищо.

Join()

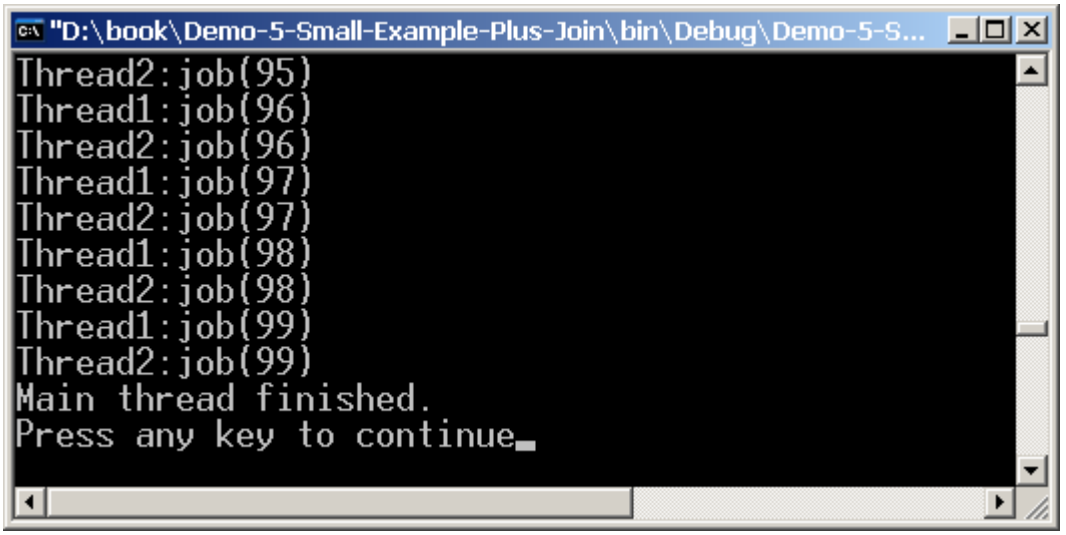
Извикващата нишка изчаква, докато извиканата приключи. Може да се укаже таймаут.

Join() – пример

Това е познатият ни пример **SmallExaple.cs**, но тук главната нишка спира работата си и продължава едва след приключването на другите две.

```
static void Main(string[] args)
{
   Console.WriteLine("Main thread started.");
   ThreadClass threadClass = new ThreadClass();
   Thread thread1 = new Thread(
      new ThreadStart(threadClass.DoTask1));
   Thread thread2 = new Thread(
      new ThreadStart(threadClass.DoTask2));
   thread1.Start();
   thread2.Start();
   thread1.Join();
   thread2.Join();
   Console.WriteLine("Main thread finished.");
}
```

Стартирането на програмата води до следния резултат:



Start()

Стартира посочената нишка. Операцията не е блокираща (връща управлението веднага). При извикване на **`Start()`** операционната система създава нова нишка и сменя състоянието ѝ в **`Running`**. При опит за повторно стартиране, се хвърля **`ThreadStateException`**.

Приоритет

В повечето имплементации на многонишковост (multithreading), се поддържа и приоритет за нишките. На базата на приоритета, планировчикът (task scheduler) определя интервала от време, който следва да бъде отделен на нишката. Операционната система не е длъжна да се съобразява с предварително зададения приоритет, но обикновено го прави.

Приоритет – пример

Ще направим нова промяна в **SmallExample.cs**. Преди да стартираме двете нишки от главната, ще променим приоритета на едната.

```
…
thread2.Priority = ThreadPriority.Highest;
thread1.Start();
thread2.Start();
…
```

Ще оставим на читателя сам да направи сравнението на резултатите.

Състояния

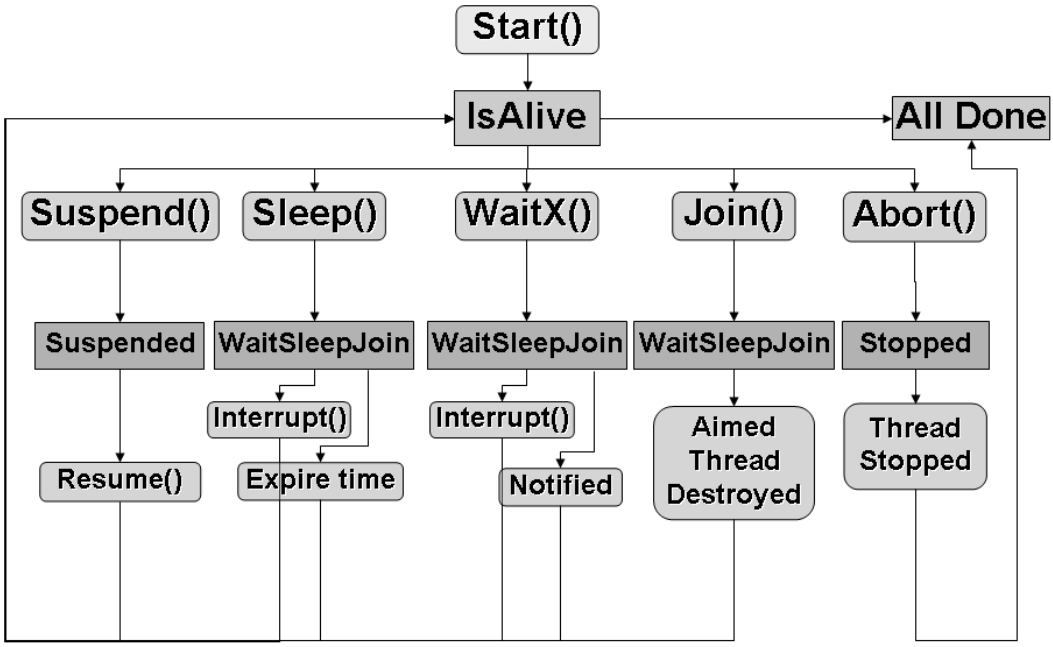
Както видяхме в примерите до момента, всяка нишка минава през различни състояния по време на своето съществуване – например да изчаква или да се изпълнява. Текущото състояние на нишката се съдържа в променливата **ThreadState**. Една нишка може да се намира и в повече от едно състояние на изброения тип **ThreadState** (понеже той има атрибут **FlagsAttribute**, който позволява побитово комбиниране на стойностите му). Отделните състояния са следните:

- **Unstarted** – нишката е създадена, но не е извикан метода **Start()**. В момента, в който **Start()** бъде извикан, нишката преминава в състояние **Running** и по никакъв начин не може да се върне обратно в това състояние.
- **Running** – нишката е стартирана, не е блокирана и не очаква да получи **ThreadAbortedException** (изключение, което се хвърля при извикване на метода **Abort()**).
- **WaitSleepJoin** – нишката е блокирана, след като е бил извикан някой от методите **Wait()**, **Sleep()** или **Join()**.
- **SuspendRequested** – за нишката е извикан метода **Suspend()**, но все още не е преустановена, а се изчаква безопасен момент това да се извърши.
- **Suspended** – нишката вече е преустановена.
- **AbortRequested** – извикан е методът **Abort()** за нишката, но тя още не е получила изключението **ThreadAbortException**, което ще се опита да я прекрати.
- **Aborted** – нишката вече е прекратена като едновременно с това се намира и в състоянието **Stopped**.
- **StopRequested** – от нишката е поискано да прекрати работата си.
- **Stopped** – нишката е прекратена или след като ѝ е бил извикан методът **Abort()**, или след като е приключила по естествен начин.

- **Background** – нишката е във фонов режим.

Живот на нишките

Съвкупността от всички състояния, през които една нишка може да премине по време на своето съществуване, определя нейния жизнен цикъл. Запознахме се със състоянията и методите, които предизвикват преходите между тях. Сега ще илюстрираме казаното със следната схема на състоянията и преходите:



Прекратяване на нишка

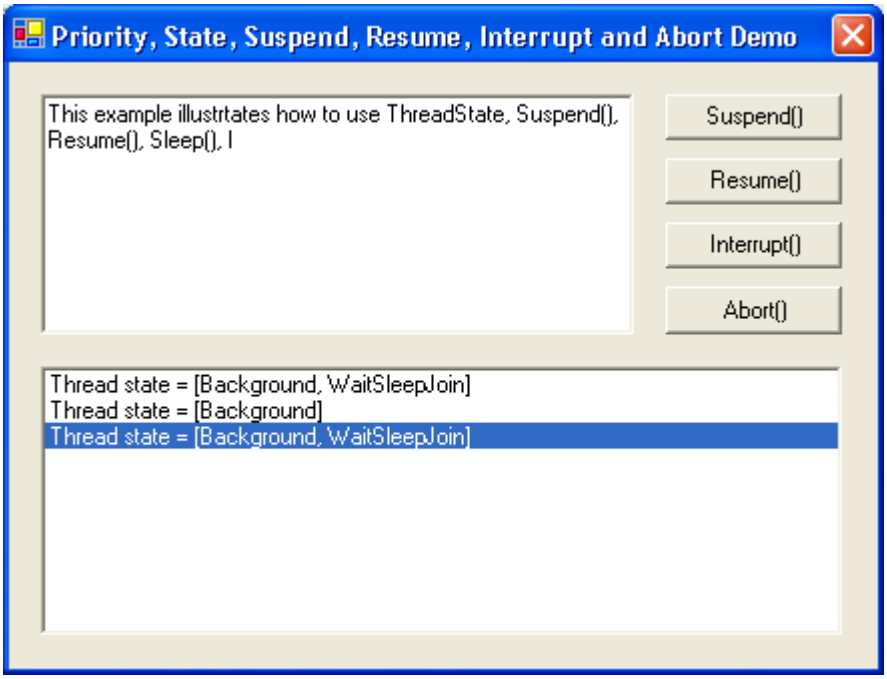
Една нишка може да бъде прекратена безусловно чрез извикване на метода **Thread.Abort()**. Извикването на този метод предизвиква **ThreadAbortedException** в нишката, за която е извикан. Това изключение е по-специално, тъй като след евентуалната си обработка в **catch** блока на нишката, то се хвърля повторно от CLR. С повторното хвърляне на изключението, CLR изпълнява всички **finally** блокове и приключва нишката. Прекратяването на нишката може да се забави неопределено дълго, в зависимост от изчисленията във **finally**, затова се препоръчва извикване на метода **Join()**, за да сме сигурни, че нишката е приключила. Повторното хвърляне на **ThreadAbortedException** може да бъде отменено чрез извикване на **Thread.ResetAbort()** в **catch** блока на прекратяваната нишка - тогава тя ще продължи изпълнението си.

Ако нишката навлезе в неуправляван код и тогава получи заявка за прекратяване, CLR "маркира" нишката и я изчаква да се върне в управляван код.

Използването на **Thread.Abort()** не е най-добрият начин да контролираме живота на една нишка. **ThreadAbortedException** е изключение, което трудно да обработено коректно. Съществуват много по-удобни механизми за синхронизация между нишки, с които ще се запознаем по-долу.

Прекратяване на нишка – пример

Ще разширим примера, който дадохме за методите **Suspend()** и **Resume()**. Програмата отново изписва текст символ по символ, но имаме възможност и да прекратим нишката във всеки момент. Паралелно с това, се следят състоянията, през които минава нишката.



Единственото, което правим в главната нишка на приложението, е да пуснем две други нишки – **mBackgroundThread**, която ще е отговорна за изписването на текста, и **mStatusWatchThread**, която ще следи състоянието на **mBackgroundThread**. И двете нишки се пускат във фонов режим.

```
private void MainForm_Load(object sender, System.EventArgs e)
{
   BackgroundThread backgroundThread =
      new BackgroundThread(this);
   mBackgroundThread = new Thread(new
      ThreadStart(backgroundThread.DoDisplayMessage));
   mBackgroundThread.IsBackground = true;
   mBackgroundThread.Start();

   StatusWatchThread statusWatchThread =
        new StatusWatchThread(this);
```

```
   mStatusWatchThread = new Thread(new
      ThreadStart(statusWatchThread.DoStatusWatch));
   mStatusWatchThread.IsBackground = true;
   mStatusWatchThread.Priority = ThreadPriority.Highest;
   mStatusWatchThread.Start();
}
```

В класа на формата са предвидени и два метода, чрез които стартираните нишки да променят графичния й интерфейс. Единият добавя нов ред в **`ListBox`** контрола, а другият присвоява текст на текстовото поле.

```
public void DisplayThreadState()
{
   string newStateMsg = String.Format("Thread state = [{0}]",
      mBackgroundThread.ThreadState);

   if (listBoxThreadState.Items.Count != 0)
   {
      string oldStateMsg = (string) listBoxThreadState.Items[
         listBoxThreadState.Items.Count-1];
      if (newStateMsg != oldStateMsg)
      {
         listBoxThreadState.Items.Add(newStateMsg);
      }
   }
   else
   {
      listBoxThreadState.Items.Add(newStateMsg);
   }

   listBoxThreadState.SelectedIndex =
      listBoxThreadState.Items.Count-1;
}

public void ShowMessageInTextBox(string aMessage)
{
   textBoxMessage.Text = aMessage;
}
```

Четирите бутона в дясно от текстовото поле викат съответните методи на **`mBackgroundThread`**. Няма да даваме тяхната имплементация.

Нишката **`mBackgroundThread`** изписва текста буква по буква и обработва възможните изключения. Отново ще подчертаем използването на **`Form.Invoke(…)`** тогава, когато потребителския интерфейс на главната нишка се променя от външна нишка.

```
delegate void StringDelegate(string aString);

public class BackgroundThread
```

```
{
   private const string MESSAGE =
   "This example illustrtates how to use ThreadState, Suspend()"+
   ", Resume(), Sleep(), Interrupt(), Abort(), Priority and "+
   "IsBackground methods and properties of the System.Threading"+
   ".Thread class.";

   private MainForm mMainForm;

   public BackgroundThread(MainForm aMainForm)
   {
      mMainForm = aMainForm;
   }

   public void DoDisplayMessage()
   {
      try
      {
         for (int len=1; len<=MESSAGE.Length; len++)
         {
            try
            {
               string msg = MESSAGE.Substring(0, len);
               mMainForm.Invoke( new
                  StringDelegate(mMainForm.ShowMessageInTextBox),
                  new object[]{msg});
            }
            catch (Exception)
            {
               return;
            }
            Thread.Sleep(100);
         }
      }
      catch (ThreadInterruptedException)
      {
         MessageBox.Show("ThreadInterruptedException", "Info");
         return;
      }
      catch (ThreadAbortException)
      {
         MessageBox.Show("ThreadAbortException", "Info");
      }
      finally
      {
         MessageBox.Show("Finally block reached.", "Info");
      }
      MessageBox.Show("Thread finished by itself.", "Info");
   }
}
```

Нишката **mStatusWatchThread** 10 пъти в секундата проверява състоянието на нишката **mBackgroundThread** и ако настъпи промяна, го отпечатва в **ListBox** контрола.

```
delegate void VoidDelegate();

public class StatusWatchThread
{
   private MainForm mMainForm;

   public StatusWatchThread(MainForm aMainForm)
   {
      mMainForm = aMainForm;
   }

   public void DoStatusWatch()
   {
      while (true)
      {
         try
         {
            mMainForm.Invoke(
               new VoidDelegate(mMainForm.DisplayThreadState));
         }
         catch (Exception)
         {
            return;
         }
         Thread.Sleep(100);
      }
   }
}
```

Как работи примерът?

Натискането на бутоните Suspend и Resume води до същия резултат, както и във вече дадения пример. Когато натиснем Abort, това предизвиква **ThreadAbortException** в **mBackgroundThread**. Изпълнява се съответния **catch** блок, след което CLR прекратява нишката, изпълнявайки преди това **finally** блока. Съобщението, което ни казва, че нишката е приключила сама, не се показва.

Нека в **catch** клаузата добавим следния ред:

```
catch (ThreadAbortException)
{
   MessageBox.Show("ThreadAbortException", "Info");
   Thread.ResetAbort();
}
```

Сега CLR не унищожава нишката, затова след обработката на изключението и **finally** блока, нишката продължава и се показва съобщението "Thread finished by itself" ("Нишката завърши сама.").

Thread Local Storage (локални за нишката данни)

Thread Local Storage е контейнер, в който всяка нишка може да съхранява собствени данни. Всеки елемент се съдържа в съответен слот за данни, който се представя от обект от класа **System.LocalDataStoreSlot**. Нишката може да си създаде такъв слот с методите **Thread.AllocateNamedDataSlot(…)** или **Thread.AllocateDataSlot()**. Ако създаденият слот е наименован, към него можем да се обръщаме и по име, в противен случай е достъпен само по референцията, върната при неговото създаване.

Слот, създаден от дадена нишка, е недостъпен за останалите нишки. Допълнително, ако в рамките на един процес е създаден слот с някакво име и друга нишка се опита да създаде нов слот със същото име, ще се хвърли изключение.

Thread Local Storage – пример

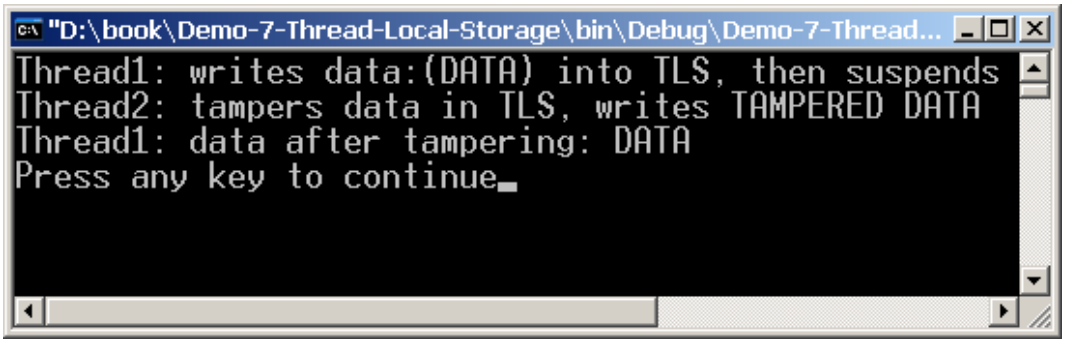
За да илюстрираме работата с Thread Local Storage ще дадем следния пример:

```
class TLSDemo
{
   [STAThread]
   static void Main(string[] args)
   {
      Threads threads = new Threads();
      Thread createDataThread = new Thread(
        new ThreadStart(threads.CreateDataThread));
      createDataThread.Start();

      Thread readDataThread = new Thread(
        new ThreadStart(threads.ReadDataThread));
      readDataThread.Start();
      readDataThread.Join();
      createDataThread.Resume();
   }
}

class Threads
{
   private const string SLOT_NAME = "temp slot";

   public void CreateDataThread()
   {
      LocalDataStoreSlot slot =
```

```
        Thread.AllocateNamedDataSlot(SLOT_NAME);
      string data = "DATA";
      Thread.SetData(slot, data);
      Console.WriteLine("Thread1: writes data:({0}) into TLS,",
        "then suspends", data);
      Thread.CurrentThread.Suspend();
      object oData = Thread.GetData(slot);
      Console.WriteLine("Thread1: data after tampering: {0}",
        oData);
    }

    public void ReadDataThread()
    {
      LocalDataStoreSlot slot =
        Thread.GetNamedDataSlot(SLOT_NAME);
      Thread.SetData(slot, "TAMPERED DATA");
      Console.WriteLine("Thread2: tampers data in TLS, writes,
        "{0}", "TAMPERED DATA");
    }
}
```

Как работи примерът?

Нишката **createDataThread** създава наименован слот за данни и записва някакви примерни данни в него. Извикването на **Suspend()** позволява на **readDataThread** да започне да се изпълнява. Тя се опитва да запише нови данни в същия слот. Тъй като този слот обаче принадлежи към Thread Local Storage на първата нишка, опитът е неуспешен и резултатът е следният:



Thread-Relative Static Fields (статични полета, свързани с нишката)

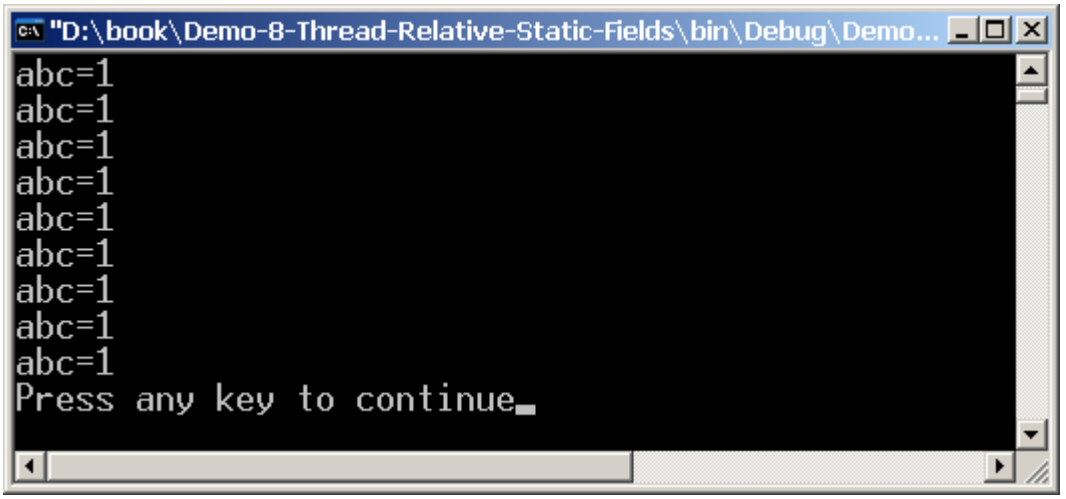
Статичните полета, свързани с нишката донякъде наподобяват обикновените статични член-променливи в един клас. Те се декларират по аналогично начин и това, което ги отличава е, че са придружени от атрибута **[ThreadStatic]**. Всяка стартирана нишка ползва отделна инстанция на тази член-променлива.

Thread-Relative Static Fields – пример

За да илюстрираме работата с атрибута **[ThreadStatic]** ще използваме следния пример:

```
class ThreadStatic
{
   [STAThread]
   static void Main(string[] args)
   {
      for( int i=0; i<10; i++ )
      {
         ThreadStart threadDelegate = new ThreadStart(new
            MyThread().DoTask);
         Thread currentThread = new Thread(threadDelegate);
         currentThread.Start();
      }
   }
}

class MyThread
{
   // This initialization is executed in the static
   // constructor, called by the main application thread
   [ThreadStatic]
   public static int abc = 42;

   public void DoTask()
   {
      abc++;
      Console.WriteLine("abc={0}", abc);
   }
}
```

Когато този кратък код се изпълни, изходът е на пръв поглед странен:



Как работи примерът?

Тук член-променливата **abc** е именно thread-relative статично поле. Десетте стартирани нишки използват десет различни инстанции на **abc**. Инициализацията **abc=42** обаче няма значение, защото конструкторът на **MyThread** се изпълнява в главната нишка. Член-променливата с атрибут [**ThreadStatic**] се инициализира отново при стартирането на нишката и нейна грижа е да го инициализира коректно.

Ако премахнем атрибута **[ThreadStatic]**, това ще бъде една обикновена статична член-променлива, обща за всички стартирани нишки:

Неудобства при работата с нишки

Не трябва да се прекалява с употребата на нишки. Управлението на много нишки и превключването от една нишка към друга отнема време, понякога надвишаващо времето за изпълнението им. От тази гледна точка, за голям брой кратки операции е добре да се използва пул от нишки (thread pool), а не много на брой нишки, които изпълняват еднократно по една малка задача.

Паралелната работа на много нишки е също трудна за следене. Тя води и до необходимостта от синхронизация, която да предотврати повреждане на данните.

Проблеми при работа с общи данни

Работата с общи данни от няколко нишки едновременно крие в себе си много опасности, които трябва да бъдат предвидени и предотвратени чрез подходящи програмни техники. Типични такива опасности са повреждането на данни (race condition) и "мъртвата хватка" (deadlock).

Повреждане на данни

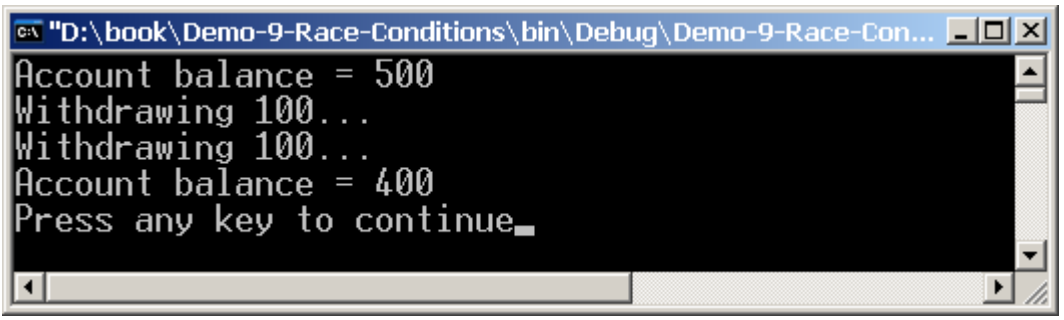
Данни, които са общи за две или повече нишки, лесно могат да бъдат повредени, ако достъпът до тях не е синхронизиран. Когато две нишки пишат едновременно в памет, заделена за някаква променлива, резултатите са непредвидими. Този проблем е известен като "повреждане на данни" или "състезание" (**race condition**).

Повреждане на данни – пример

За пример ще дадем един клас, който представлява банкова сметка. Когато две нишки едновременно теглят пари от тази банкова сметка, остатъкът в нея става некоректен.

```
class Bank
{
   static void Main(string[] args)
   {
      Account acc = new Account();
      acc.mBalance = 500;
      Console.WriteLine("Account balance = {0}", acc.mBalance);
      Thread user1 = new Thread( new ThreadStart
         (acc.Withdraw100) );
      Thread user2 = new Thread( new ThreadStart
         (acc.Withdraw100) );
      user1.Start();
      user2.Start();
      user1.Join();
      user2.Join();
      Console.WriteLine("Account balance = {0}", acc.mBalance);
   }
}

class Account
{
   public int mBalance;

   public void Withdraw100()
   {
      int oldBalance = mBalance;
      Console.WriteLine("Withdrawing 100...");
      // Simulate some delay during the processing
      Thread.Sleep(100);
      int newBalance = oldBalance - 100;
      mBalance = newBalance;
   }
}
```

След изпълнението на програмата, остатъкът по сметката не е 300, а 400:



Резултатът е изненадващ, защото двете нишки едновременно променят една и съща член-променлива. Получената грешка е времезависима – ако приспим нишките за друг интервал от време, или пък не ги приспим изобщо, резултатът може и да е верен.

Мъртва хватка (deadlock)

Друг опасен синхронизационен проблем е т.нар. "мъртва хватка" (**deadlock**). Това е състояние, при което две нишки взаимно се чакат за освобождаване на заети от тях ресурси. Например нишка A използва ресурса X и би го освободила при възможност да заеме ресурс Y. Нишка B, от своя страна, използва Y и чака X. Получава се "увисване", при което нито една от двете нишки не може да продължи.

Типично за ситуацията "мъртва хватка" е, че не може да се получи, ако споделеният ресурс е само един. Ако ресурсите са няколко, "мъртвата хватка" може да се избегне, ако те се взимат в еднакъв ред от различните нишки. Например ако в предишния пример нишка A първо взима ресурса X, а след това Y и нишка B се опитва да вземе в същия ред първо ресурс X, а след това ресурс Y, не може да се получи безкрайно чакане. Или нишката ще вземе двата ресурса, или нишката B – според това коя е била първа.

Синхронизация

В края на предишната точка показахме до какво може да доведе едновременният достъп до общи ресурси. Целта на синхронизацията е това да не се допуска. Тук ще разгледаме някои стратегии за синхронизация.

Най-доброто решение за общите данни

В идеалния вариант, изобщо нямаме споделени данни. Ако данните в обектите са капсулирани така, че да не е нужно да бъдат споделяни между две и повече нишки, проблемите с общите данни автоматично отпадат. Понякога обаче е наложително да споделяме данни и в такъв случай трябва да използваме механизмите за синхронизация, които .NET Framework предлага.

Синхронизирани "пасажи" код (synchronized code regions)

Тук се синхронизират само отделни участъци от кода – тези, които са рискови. Критична секция наричаме участък от кода, до който не трябва да бъде допускан едновременен достъп. За гарантиране безопасен достъп до критична секция може да използваме ключовата дума **lock** или класа **Monitor**.

```
lock (obj)
{
   // code
}
```

или

```
Monitor.Enter(obj);
try
{
   // code
}
finally
{
   Monitor.Exit(obj);
}
```

Обектът **obj** трябва да бъде от референтен тип (ако не е, се извършва опаковане, което ще доведе до безрезултатно заключване на различен новосъздаден обект при всяко влизане в секцията). На мястото на **obj** да често се ползва **this** или член-променлива, дефинирана специално за целта. В случаите, когато искаме да защитим статична член-променлива или критичната секция е в тялото на статичен метод, **obj** може да бъде **typeof(class)**.

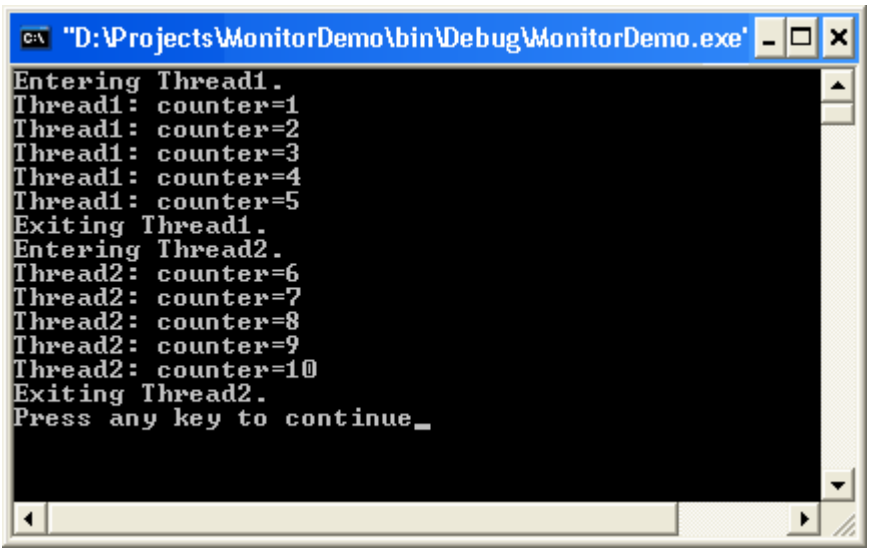
Работа с критични секции – пример

От главната нишка на програмата ще пуснем две други нишки, стартиращи от един и същ метод. Когато едната нишка започне изпълнение, другата ще чака, защото обработката на двете нишки е в критична секция и достъпът до нея е синхронизиран.

```
public class MonitorEnterExitDemo
{
   private int mCounter = 0;

   public void CriticalSection()
   {
      Monitor.Enter(this);
      try
```

```
      {
         Console.WriteLine("Entering {0}.",
            Thread.CurrentThread.Name);

         for(int i = 1; i <= 5; i++)
         {
            mCounter++;
            Console.WriteLine("{0}: counter={1}",
               Thread.CurrentThread.Name, mCounter);
            Thread.Sleep(500);
         }

         Console.WriteLine("Exiting {0}.",
            Thread.CurrentThread.Name);
      }
      finally
      {
         Monitor.Exit(this);
      }
   }

   public static void Main()
   {
      MonitorEnterExitDemo demo = new MonitorEnterExitDemo();

      Thread thread1 = new Thread(new
         ThreadStart(demo.CriticalSection));
      thread1.Name = "Thread1";
      thread1.Start();

      Thread thread2 = new Thread(new
         ThreadStart(demo.CriticalSection));
      thread2.Name = "Thread2";
      thread2.Start();
   }
}
```

Когато изпълним програмата, виждаме, че втората нишка влиза в критичната си секция едва след като първата е приключила:



Как работи примерът?

Изразът **`Monitor.Enter(this)`** поставя началото на критичната секция. Нишката, която първа го изпълни (в случая, това е **`thread1`**), "заключва" кода след този ред до освобождаването на монитора с **`Monitor.Exit(this);`** във **`finally`** клаузата. Едва тогава, след като критичната секция е "отключена", другата нишка може да влезе в нея.

Същият ефект може да се постигне и с ключовата дума **`lock`**.

Ще оставим на читателя сам да направи сравнението при липса на синхронизация.

Методите Wait(...) и Pulse(...)

`Wait(object)` и **`Pulse(object)`** са два от важните методи на класа **`Monitor`**. Извикването на **`Monitor.Wait(object)`** освобождава монитора на посочения обект и блокира викащата нишка, докато не си върне монитора. Това блокиране трае до извикването на **`Monitor.Pulse(object)`** от друга нишка. При блокирането на нишката може да се укаже таймаут. Ако такъв няма, нишката може да остане блокирана завинаги, в случай, че **`Pulse(…)`** не бъде извикан. В този интервал от време, нишката стои в опашката на чакащи нишки.

Методът **`Monitor.Pulse(…)`** може да се извика само от текущия притежател на монитора на обекта – т.е. от критична секция. Нишката преминава в опашката на нишки, готови да се изпълняват и да вземат монитора.

Към тези два метода можем да причислим и **`Monitor.PulseAll(…)`**, който има действие, аналогично на **`Pulse(…)`**, но за цялата опашка от чакащи нишки.

Wait(…) и Pulse(…) – пример

Демонстрацията илюстрира синхронизация между нишки чрез заспиване и събуждане (**`Monitor.Wait(…)`** и **`Monitor.Pulse(…)`**). В примера се създават две нишки, всяка от които извършва някаква работа, събужда другата и заспива.

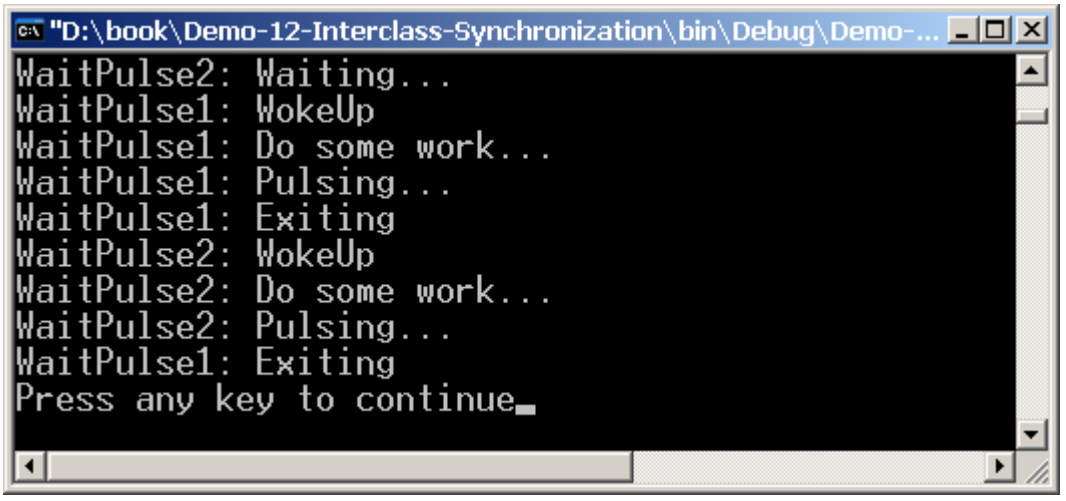
```
public class WaitPulse
{
   private object mSync;
   private string mName;

   public WaitPulse(string aName, object aSync)
   {
      mName = aName;
      mSync = aSync;
   }

   public void DoJob()
   {
      lock (mSync)
      {
         Console.WriteLine("{0}: Start", mName);

         Console.WriteLine("{0}: Pulsing...", mName);
         Monitor.Pulse(mSync);

         for(int i = 1; i <= 3; i++)
         {
            Console.WriteLine("{0}: Waiting...", mName);
            Monitor.Wait(mSync);

            Console.WriteLine("{0}: WokeUp", mName);
            Console.WriteLine("{0}: Do some work...", mName);
            Thread.Sleep(1000);

            Console.WriteLine("{0}: Pulsing...", mName);
            Monitor.Pulse(mSync);
         }
         Console.WriteLine("{0}: Exiting", mName);
      }
   }
}

public class WaitPulseDemo
{
   public static void Main(String[] args)
   {
      object sync = new object();

      WaitPulse wp1 = new WaitPulse("WaitPulse1", sync);
```

```
        Thread thread1 = new Thread(new ThreadStart(wp1.DoJob));
        thread1.Start();

        WaitPulse wp2 = new WaitPulse("WaitPulse2", sync);
        Thread thread2 = new Thread(new ThreadStart(wp2.DoJob));
        thread2.Start();
    }
}
```

Как работи примерът?

При стартиране, създаваме обекта **sync**. Когато създаваме нишките, им предаваме този обект и синхронизацията се извършва по него. Всяка от нишките извиква **Monitor.Pulse(mSync)**, с което събужда другата нишка, ако тя е заспала. След това, в цикъл, всяка от нишките заспива, докато не бъде събудена от другата, върши някаква работа и събужда другата.

В резултат, двете нишки се редуват – докато едната работи, другата спи.



Синхронизирани контексти (Synchronized Contexts)

Това е синхронизация на ниво клас. За целта, класът трябва да наследява **ContextBoundObject**. Обектите от такъв клас оперират в един контекста, който е част от домейна на приложението. Ако за такъв клас използваме атрибута **SynchronizationAttribute**, неговите методи са нишково обезопасени, т. е. два или повече метода не могат да бъдат изпълнявани едновременно от различни нишки. Статичните членове обаче не са предпазени.

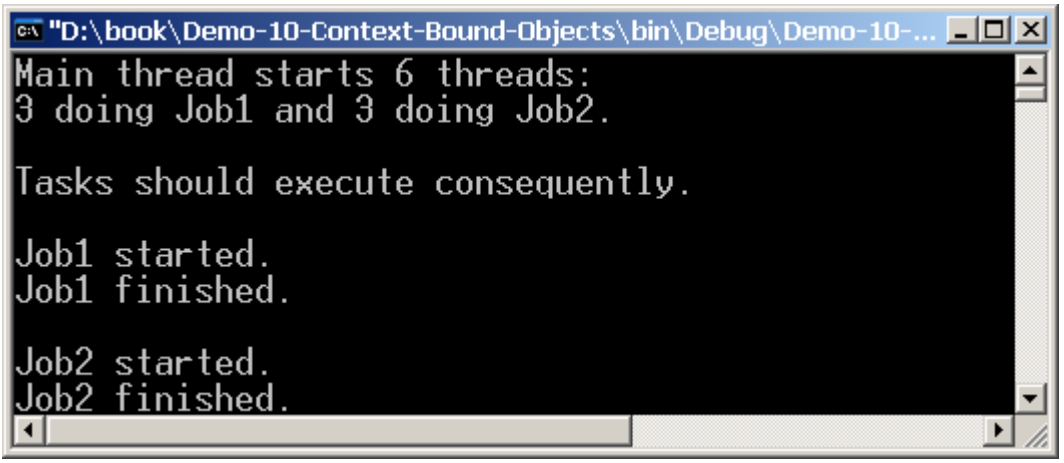
Синхронизирането е на ниво клас – не можем да поддържаме синхронизация по отношение на някакъв участък от кода.

Синхронизиран контекст – пример

Класът **CBO** е наследник на **ContextBoundObject** и има атрибут **[SynchronizationAttribute]**. Два негови метода служат за тяло на общо 6 нишки. Единият метод е по-бърз от другия, като това не влияе върху синхронизацията.

```
class Starter
{
   static void Main()
   {
      CBO syncClass = new CBO();
      Console.WriteLine("Main thread starts 6 threads:\n" +
         "3 doing Job1 and 3 doing Job2.\n\n" +
         "Tasks should execute consequently.\n");
      for (int i=0; i<6; i++)
      {
         Thread t;
         if( i%2==0 )
            t = new Thread( new ThreadStart(
               syncClass.DoSomeTask1) );
         else
            t = new Thread( new ThreadStart(
               syncClass.DoSomeTask2) );
         t.Start();
      }
   }
}

[SynchronizationAttribute]
class CBO : ContextBoundObject
{
   public void DoSomeTask1()
   {
      Console.WriteLine("Job1 started.");
      Thread.Sleep(2000);
      Console.WriteLine("Job1 finished.\n");
   }

   public void DoSomeTask2()
   {
      Console.WriteLine("Job2 started.");
      Thread.Sleep(1500);
      Console.WriteLine("Job2 finished.\n");
   }
}
```

Резултатът е следният:



В даден момент, не повече от един метод на класа може да се изпълнява и нишките се изчакват една друга.

MethodImplAttribute

`MethodImplAttribute` е атрибут, позволяващ "заключване" на цял метод, независимо от това дали методът е статичен или не. Използва се по следния начин:

```
[MethodImpl(MethodImplOptions.Synchronized)]
public void DoSomeTask()
{
   // Some code
}
```

По този начин може да синхронизираме достъпа до **`DoSomeTask()`**. Аналогичен ще бъде резултатът, ако използваме ключовата дума **`lock`** върху кода на целия метод.

Неуправлявана синхронизация – класът WaitHandle

Синхронизацията, която разгледахме до момента, беше управлявана синхронизация. Винаги, когато използваме ключовата дума **`lock`**, класа **`Monitor`** или атрибути за синхронизация, това е синхронизация, контролирана от CLR.

В тази точка ще слезем на малко по-ниско ниво, за да разгледаме неуправляваната синхронизация (unmanaged synchronization). При нея се използват обекти на операционната система.

Неуправляваната синхронизация в .NET Framework е представена от базовия клас **`WaitHandle`** и неговите наследници – **`Mutex`**, **`AutoResetEvent`** и **`ManualResetEvent`**. Обектите от тези класове са примитиви за синхронизация на операционната система. Методите на **`WaitHandle`** се използват

за изчакването на събития. Синхронизацията се основава на "сигнализирането" на тези събития.

Добре е да се внимава с употребата на неуправлявана синхронизация. Независимо, че на моменти тя дава по-големи възможности от управляваната, нейната зависимост от операционната система прави преносимостта на кода по-трудна. Освен това, класът **Monitor** използва по-ефективно системните ресурси.

WaitHandle – по-важни методи

Ето някои от най-често използваните методи на класа **WaitHandle**:

- **static bool WaitAny(WaitHandle[])**
- **static bool WaitAll(WaitHandle[])**
- **virtual bool WaitOne()**

Трите изброени метода са предефинирани в класа **WaitHandle**, но за да обясним действието им ще се спрем само на този техен базов формат.

WaitAny(…) блокира текущата нишка до получаването на първия сигнал от масив от **WaitHandle** обекти, а **WaitAll(…)** – до получаване на сигнал от всички обекти. Тези методи са без аналог при управляваната синхронизация, напр. чрез класа **Monitor**.

За разлика от първите два метода, които са статични, **WaitOne()** е метод на инстанцията. Когато се предефинира в клас, наследник на **WaitHandle**, той блокира текущата нишка, докато текущия **WaitHandle** обект получи сигнал. В следващата точка ще демонстрираме употребата на този метод за класа **Mutex**.

Класът Mutex

Класът **Mutex** е наследник на **WaitHandle** и представлява "мутекс" - примитив за синхронизация на операционната система. Той наподобява **Monitor**, но не е свързан с обект. Самата дума "мутекс" произлиза от английския термин за взаимно изключване (mutual exclusion).

Когато една нишка придобие мутекса, друга може да го вземе едва след като първата го освободи. Всяка нишка може да поиска мутекса с **Mutex.WaitOne()** и след като приключи работата си, да го освободи с **Mutex.ReleaseMutex()**. Веднъж придобила мутекс чрез извикване на **WaitOne()**, нишката може да извика същия метод произволен брой пъти, като продължава нормалното си изпълнение. За да бъде освободен мутекса обаче, **ReleaseMutex()** трябва да бъде извикан същия брой пъти.

Методите **WaitAll(…)** и **WaitAny(…)**, дефинирани в базовия клас **WaitHandle**, тук могат успешно да се прилагат.

Синхронизация с Mutex – пример

Следващият код решава познатата задача за синхронизиран достъп до дадена критична секция, но чрез **`Mutex`** обект.

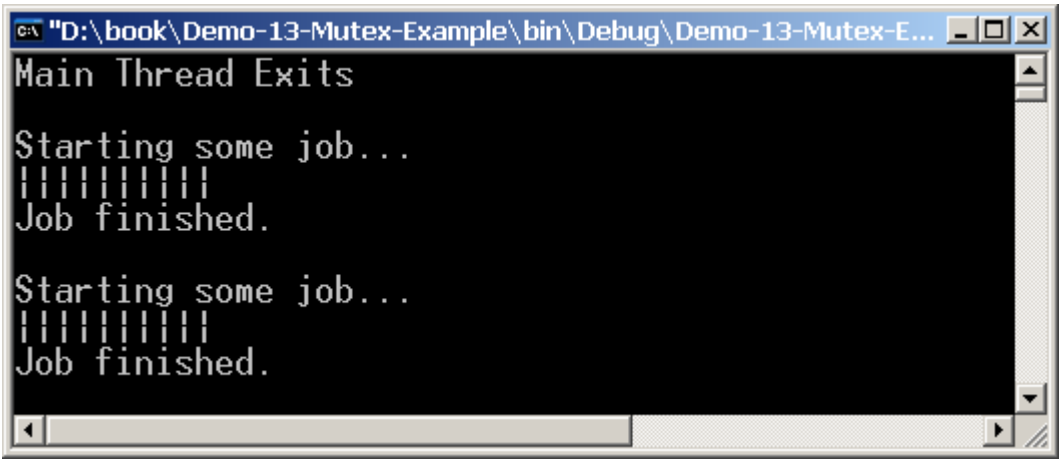
```
class MutexMain
{
   const int THREAD_COUNT = 5;

   static void Main(string[] args)
   {
      Mutex commonMutex = new Mutex();
      Thread[] demoThreads = new Thread[THREAD_COUNT];
      for (int i=0; i<THREAD_COUNT; i++)
      {
         MutexThread mutexThread = new MutexThread(commonMutex);
         demoThreads[i] = new Thread(
            new ThreadStart(mutexThread.PerformSomeTask));
         demoThreads[i].Start();
      }

      foreach (Thread thread in demoThreads)
      {
         thread.Join();
      }

      Console.WriteLine("Main Thread Exits");
   }
}

class MutexThread
{
   Mutex mMutex;

   public MutexThread(Mutex aMutex)
   {
      mMutex = aMutex;
   }

   public void PerformSomeTask()
   {
      mMutex.WaitOne();
      Thread.Sleep(200);
      Console.WriteLine("\nStarting some job...");
      for (int i=0; i<10; i++)
      {
         Thread.Sleep(100);
         Console.Write("|");
      }
      Console.WriteLine("\nJob finished.");
      mMutex.ReleaseMutex();
```

```
    }
}
```

Как работи примерът?

На всички нишки, които създаваме в метода **Main(…)**, подаваме един обект от клас **Mutex**. Така всички нишки от масива **demoThreads** работят с един и същ мутекс. Всички нишки имат за обработка метода **PerformSomeTask()**. Когато някоя от стартираните нишки изпълни реда **mMutex.WaitOne();**, тя получава мутекса ако е свободен, влиза в критичната секция, свършва някаква работа и освобождава мутекса с **mMutex.ReleaseMutex()**. Така се гарантира взаимното изключване.

Резултатът от изпълнението е следният:



Класовете AutoResetEvent и ManualResetEvent

Това са още два класа, които наследяват **WaitHandle** и представляват примитиви за синхронизация. Обектите от клас **AutoResetEvent** и **ManualResetEvent** са събития и могат да имат две състояния – сигнализирани и несигнализирани. Едно събитие може явно да се установи в сигнализирано състояние с метода **Set()** и в несигнализирано – с **Reset()**.

AutoResetEvent обект, сигнализиран чрез **Set()**, сигнализира само първия чакащ манипулатор. След първия изпълнен **WaitOne()** от този обект, събитието се връща в несигнализирано състояние. Ако събитието обаче е от клас **ManualResetEvent**, то сигнализира всички чакащи манипулатори. Веднъж сигнализирано, то може да бъде върнато в несигнализирано състояние единствено с извикване на **Reset()**.

AutoResetEvent и ManualResetEvent – пример

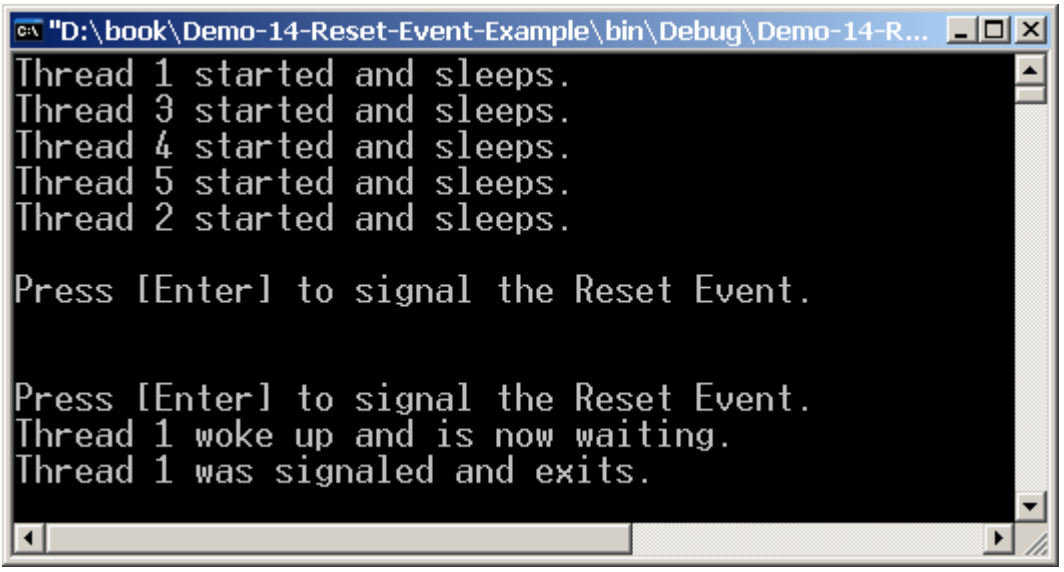
Със следващия пример ще демонстрираме работата с класовете **AutoResetEvent** и **ManualResetEvent** и ще покажем разликите при сигнализирането на събитията. Нека най-напред разгледаме случая, в който събитието е от тип **AutoResetEvent**.

```
class MainClass
{
   const int THREADS_COUNT = 5;

   static void Main()
   {
      AutoResetEvent evnt = new AutoResetEvent(false);

      for (int i=0; i<THREADS_COUNT; i++)
      {
         OneWhoWaits oww = new OneWhoWaits(evnt, (i+1)*500);
         Thread thread = new Thread(new
            ThreadStart(oww.PerformSomeTask));
         thread.Start();
      }

      Thread.Sleep(100);

      for (int i=0; i<THREADS_COUNT; i++)
      {
         Console.WriteLine("\nPress [Enter] to signal the Reset"+
            " Event.");
         Console.ReadLine();
         evnt.Set();
      }

      Console.WriteLine("\nMain thread finished.");
   }
}

class OneWhoWaits
{
   WaitHandle mWaitHandle;
   int mWaitTime;

   public OneWhoWaits(WaitHandle aWaitHandle, int aWaitTime)
   {
      mWaitHandle = aWaitHandle;
      mWaitTime = aWaitTime;
   }

   public void PerformSomeTask()
   {
      Console.WriteLine("Thread {0} started and sleeps.",
```

```
            Thread.CurrentThread.GetHashCode());
        Thread.Sleep(mWaitTime);
        Console.WriteLine("Thread {0} woke up and is now waiting.",
            Thread.CurrentThread.GetHashCode());
        mWaitHandle.WaitOne();
        Console.WriteLine("Thread {0} was signaled and exits.",
            Thread.CurrentThread.GetHashCode());
    }
}
```

Как работи примерът?

Най-напред, създаваме синхронизационния обект **evnt** и го подаваме на петте нишки, които стартираме. Аргументът **false** в конструктора на **evnt** показва, че събитието е в несигнализирано състояние при създаването си. Стартираните нишки се блокират на реда **mWaitHandle.WaitOne();** и чакат потребителя да натисне [Enter], с което да се сигнализира събитието. Тъй като събитието е от тип **AutoResetEvent**, с всяко натискане на [Enter] пропускаме по една нишка. След петото натискане, всички нишки приключват.



Нека сменим само типа на събитието, което създаваме:

```
ManualResetEvent evnt = new ManualResetEvent(false);
```

Сега първото натискане на [Enter] води до приключване на всички нишки – включително и тези, които още не са започнали да чакат. Това е така, защото след реда **evnt.Set();**, събитието никъде не се връща в несигнализирано състояние. **ManualResetEvent** събитието може да се

върне в несигнализирано състояние само с **Reset()**, затова нека направим и тази промяна:

```
for (int i=0; i<THREADS_COUNT; i++)
{
   Console.WriteLine("\nPress [Enter] to signal the Reset "+
      "Event.");
   Console.ReadLine();
   evnt.Set();
   // code added
   Thread.Sleep(10);
   evnt.Reset();
}
```

Сега натискането на [Enter] предизвиква пропускане само на нишките, които в този момент са чакащи – достигналите до реда **mWaitHandle.WaitOne();**. Тъй като след сигнализирането на събитието, го връщаме ръчно в несигнализирано състояние, за останалите нишки то вече е несигнализирано и те чакат ново натискане на [Enter].

Класът Interlocked

Понякога единственото, което ни трябва, е да увеличим или намалим дадена стойност или да разменим стойности по синхронизиран начин. Разбира се, можем за целта да използваме мутекси, но това до голяма степен ще усложни кода ни. За удовлетворяване на тези често срещани изисквания .NET Framework предоставя класа **Interlocked**. Той предлага няколко статични метода за атомарна работа с променливи. Атомарна наричаме всяка операция, която или се изпълнява цялата, или не се изпълнява изобщо.

Increment(…) и Decrement(…)

Методите **Increment(…)** и **Decrement(…)** служат съответно за увеличаване и намаляване на стойност. Те приемат единствен параметър от тип **ref int** или **ref long** и като резултат връщат стойността, получена след извършване на операцията.

```
int i = 2;
int newValue = Interlocked.Increment(ref i);
Debug.Assert(i == 3);
Debug.Assert(newValue == 3);
```

Ако увеличим променливата с **i++**, това не е атомарна операция – стойността на променливата се записва в регистър, стойността ѝ се увеличава и се записва обратно в променливата, или общо три операции.

Exchange(...) и CompareExchange(...)

Методът **Exchange(…)** служи за размяна на стойности, докато **CompareExchange(…)** сравнява две променливи и ако са равни по стойност, указва нова стойност за едната. Двата метода имат по три версии, различаващи се само в типа на параметрите, с които оперират (**int**, **float** или **object**). Връщаният резултат е от тип, същия като типа на аргументите им.

Пример с CompareExchange(...)

Докато предназначението на метода **Exchange(…)** е ясно, то семантиката на **CompareExchange(…)** не е толкова проста и затова ще илюстрираме действието му с пример:

```
using System.Threading;

public class ThreadSafeTotalAccumulation
{
   private int totalValue = 0;

   public int AddToTotal(int valueToAdd)
   {
      int initialValue, computedValue;

      do
      {
         initialValue = totalValue;
         computedValue = initialValue + valueToAdd;

      } while (initialValue != Interlocked.CompareExchange(
            ref totalValue, computedValue, initialValue));

      return computedValue;
   }
}
```

Как работи примерът?

Класът **ThreadSafeTotalAccumulation** съдържа поле **totalValue**, към което искаме да добавим някаква стойност по нишково безопасен начин. Когато влезем в цикъла, запомняме старата сума в **initialValue** и пресмятаме новата в **computedValue**. **CompareExchange(…)** сравнява **totalValue** и **initialValue**. Ако не са равни, значи друга нишка е успяла да обнови общата сума по време на изпълнение на цикъла. Тогава **CompareExchange(…)** не обновява **totalValue**, а връща съдържанието на **totalValue**, което е различно от **initialValue**, и цикълът се повтаря. В момента на излизане от цикъла, **computedValue** е записан в **totalValue**.

Връщаме **computedValue**, а не **totalValue**, защото **totalValue** може междувременно да бъде променена.

Класът **Interlocked** е полезен само в случаите, когато промяната на променливите минава винаги през него и никога не ги модифицираме директно.

Класически синхронизационни задачи

Случаите, в които две и повече нишки се конкурират за общи ресурси, често си приличат. Известни са няколко основни категории проблеми, представени от следните класически синхронизационни задачи

Задачата "Производител - потребител" (The Producer – Consumer Problem)

Две нишки, условно наречени "производител" и "потребител", споделят обща опашка от данни с някаква дължина. Производителят създава данни и ги прибавя към опашката. От своя страна, потребителят ги чете от нея. Проблемите, които възникват, са следните:

- Поради ограничения размер на опашката, производителят не трябва да се опитва да записва данни в нея, когато е пълна. Ако това е така, той чака, докато потребителят прочете някой от елементите и освободи място.
- Потребителят не трябва да се опитва да чете от празна опашка. В този случай той ще чака, докато производителят добави нов елемент.

Задачата е известна още под името "ограничен буфер" (bounded buffer). Нейният частен случай, в който дължината на опашката е безкрайна, е известна като "неограничен буфер" (unbounded buffer). Тогава отпада условието производителят да не пише в пълна опашка и решението се опростява.

Задачата "Производител - потребител" – примерно решение

В .NET Framework не е предоставен стандартен клас за решение на този проблем, но приложението на тази задача в практиката е голямо. Ще дадем примерно решение на проблема, което лесно позволява да бъде преизползвано при нужда:

```
using System;
using System.Collections;
using System.Threading;

public class SharedQueue
{
  private static object[] mSharedQueue;
  private static int mCurrentElementPointer = -1;
```

```
        private static int mCapacity;

        public SharedQueue(int aCapacity)
        {
          mSharedQueue = new object[aCapacity];
          mCapacity = aCapacity;
        }

        public void Enqueue(object aObject)
        {
          while(true)
          {
            lock(this)
            {
              if(mCurrentElementPointer < mCapacity-1)
              {
                mCurrentElementPointer++;
                mSharedQueue[mCurrentElementPointer] =
aObject;
                Monitor.Pulse(this);
                return;
              }
              else
              {
                Monitor.Wait(this);
              }
            }
          }
        }

        public object Dequeue()
        {
          while(true)
          {
            lock(this)
            {
              if(mCurrentElementPointer != -1)
              {
                object result=
mSharedQueue[mCurrentElementPointer];
                mCurrentElementPointer--;
                Monitor.Pulse(this);
                return result;
              }
              else
              {
                Monitor.Wait(this);
              }
            }
          }
```

```
            }
        }
```

Как работи примерът?

Реализиран е случаят с ограничен буфер. Операциите добавяне и изваждане на елемент са синхронизирани и блокират съответно при препълнена или празна опашка.

Добавянето на елемент в опашката (вж. метода **`Enqueue(…)`**) е възможно само когато никой не я ползва в дадения момент и тя не е препълнена. Ако в момента опашката се ползва (т.е. е заключена), чакаме да бъде отключена. Това се осигурява от **`lock`** блока. След това, ако опашката не е препълнена, добавяме новия елемент в нея и викаме **`Monitor.Pulse()`**, за да събудим чакащите нишки, блокирали в метода **`Dequeue()`** (ако има такива). Ако опашката е препълнена, приспиваме с **`Monitor.Wait()`** текущата нишка. Тя ще бъде събудена от друга нишка, която успешно е изпълнила метода **`Dequeue()`** и е освободила място в опашката.

Изваждането на елемент от опашката работи абсолютно аналогично на добавянето.

Задачата "Четци и писачи" (The Readers-Writers Problem)

В тази задача имаме един или повече "писачи", които искат да пишат върху даден общ ресурс, например файл. Успоредно на тях, един или повече "четци" четат от същия ресурс. За да е коректен достъпът до общия ресурс, необходимо е да са спазени следващите условия (условия на Бернщайн):

- Произволен брой четци могат да имат едновременен достъп до ресурса – това няма как да породи синхронизационни проблеми, защото в този момент ресурсът не се променя.
- Ако на писач е предоставен достъп до ресурса, достъпът на всички останали трябва да бъде забранен – независимо дали четци или писачи.
- Нито един четец или писач не трябва да чака безкрайно дълго

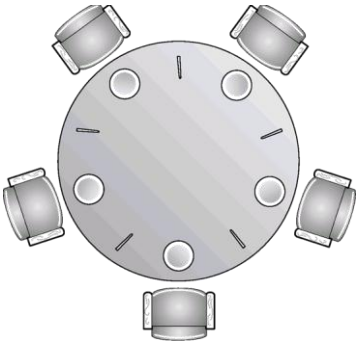
.NET Framework предлага решение на тази задача - класът **`ReaderWriterLock`**. Критичният ресурс се заключва с методите **`AcquireReaderLock(…)`** и **`AcquireWriterLock(…)`**, съответно за четец и писач. Освобождаването става с **`ReleaseReaderLock()`** и **`ReleaseWriterLock()`**. Свойствата **`IsReaderLockHeld`** и **`IsWriterLockHeld`** ни информират дали ресурсът е текущо заключен от четец или от писач.

Ето един примерен алгоритъм за това, как да използваме класа **`ReaderWriterLock`**.

```
class Resource
```

```
{
   ReaderWriterLock rwLock = new ReaderWriterLock();

   public void Read()
   {
      rwLock.AcquireReaderLock(Timeout.Infinite);
      try
      {
         // Many can read, writers are blocked
      }
      finally
      {
         rwLock.ReleaseReaderLock();
      }
   }

   public void Write()
   {
      rwLock.AcquireWriterLock(Timeout.Infinite);
      try
      {
         // One can write, readers are blocked
      }
      finally
      {
         rwLock.ReleaseWriterLock();
      }
   }
}
```

Задачата "Обядващи философи" (The Dining Philosophers Problem)



В тази задача, няколко философа стоят около кръгла маса и всеки от тях извършва само 2 действия – храни се или мисли. За да започне даден философ да се храни, той се нуждае едновременно от двете вилици, които стоят вляво и вдясно от чинията му. Ако един философ вземе едната вилица, но не може да вземе в този момент и другата (защото тя е заета), той не може да започне да се храни докато не се сдобие и с нея. Има риск

всеки философ да хване една от вилиците в даден момент и да чака безкрайно за другата. Това ще доведе до "мъртва хватка" (deadlock). Задачата е да се измисли алгоритъм за хранене на философите, при който не се получават "мъртви хватки".

Задачата "Обядващи философи" – решения

Едно примерно решение на проблема е да наредим вилиците и да изискваме философите да ги вземат в нарастващ ред. Нека имаме 5 философа обозначени с P1, P2, P3, P4, и P5, а вилиците да са номерирани с F1, F2, F3, F4, и F5. Първият философ (P1) ще вземе първата вилица (F1) преди да се посегне към втората (F2). Философите от P2 до P4 ще се държат аналогично, вземайки Fx преди Fx+1. Философът P5 обаче ще вземе F1 преди F5 и именно тази асиметрия ще предотврати "мъртва хватка". Имплементацията на това решение е тривиална.

Друго просто решение на проблема е да разгледаме масата като споделен ресурс и при започване на операцията "взимане на две вилици" да използваме заключване на масата с критична секция. Аналогично постъпваме и при операцията "връщане на две вилици". По този начин правим операциите "взимане на двете вилици" и "връщане на двете вилици" атомарни, а това означава, че не може да се получи "мъртва хватка".

Пул от нишки (ThreadPool)

През голям период от своето съществуване, нишката се намира в състояние **`ThreadState.WaitSleepJoin`** – очакваща случването на някакво събитие или приспана със **`Sleep(…)`**. Понякога нишката се "събужда" за много кратки периоди, само за да провери дали е изпълнено някакво условие. Поддържането на много неактивни нишки е излишно и консумира ресурси.

Подходът на пула от нишки намалява натоварването при създаване и унищожаване на нишки. Група нишки, наречени работни нишки (worker threads), се създават в началото на многонишковото приложение и формират пул. Работните нишки са фиксиран брой – веднъж създадени, не се убиват и не се създават нови. При нова задача, пулът предоставя работна нишка за нейното изпълнение. След приключване на работата, нишката се връща в пула без да се унищожава. Механизмът е подходящ за много на брой задачи, които могат да се изпълняват паралелно. Задачите за изпълнение се нареждат в опашка и започват да се изпълняват при предоставена им работна нишка.

Един процес може да има само един пул от нишки, общ за всички домейни на приложението в процеса. Стандартно, пулът от нишки е ограничен на 25 нишки на процесор.

Предимства

Пулът от нишки преизползва нишките. Не се губи време за създаване и унищожаване на нишки.

Задачата, обслужвана от работните нишки, се освобождава от задължението да ги създава и контролира.

Увеличаването на производителността е не само по отношение на текущото приложение, но и по отношение на другите стартирани процеси. Постоянният брой на работните нишки позволява на операционната система да оптимизира кванта от време, предоставян на нишките от всички процеси.

Недостатъци

Пулът от нишки е неудобен, когато е нужна контролираща нишка. Всички работни нишки са равнопоставени.

Работните нишки не трябва да работят върху споделени данни. Ако има нужда от синхронизация, пулът не е добро решение, защото по своята същност е асинхронен.

Ако някоя от задачите отнема много време, тя може да забави останалите.

Ако дадена задача е в пула от нишки, тя не може да се премахне от него.

Класът ThreadPool

В .NET Framework, пулът от нишки е имплементиран в класа **`ThreadPool`**. Чрез метода **`QueueUserWorkItem(…)`** добавяме нова задача в опашката. Първото извикване на метода създава пула от нишки на процеса.

ThreadPool – пример

Ще дадем следния пример за добавяне на задачи в опашката на пула от нишки и тяхното изпълнение:

```
class ThreadPoolDemo
{
   const int TASKS_COUNT = 100;

   public static void LongTask(object aParam)
   {
      Console.WriteLine("Started: {0}.", aParam);
      Thread.Sleep(500);
      Console.WriteLine("Finished: {0}.", aParam);
   }

   static void Main()
   {
      Console.WriteLine("Press [Enter] to exit.");
```

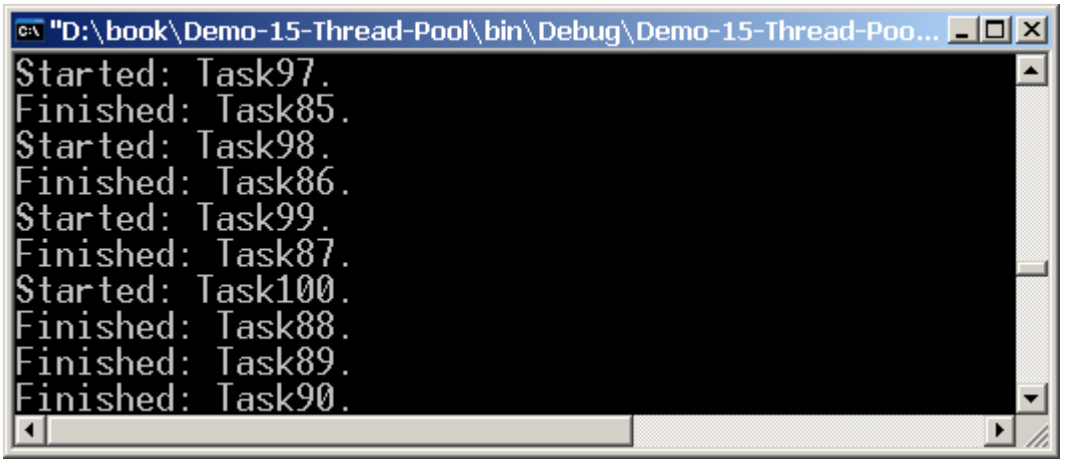
```
      for (int i=1; i<=TASKS_COUNT; i++)
      {
         string taskName = "Task" + i;
         ThreadPool.QueueUserWorkItem(new
            WaitCallback(LongTask), taskName);
      }

      Console.ReadLine();
   }
}
```

Как работи примерът?

Най-напред, главната нишка на приложението добавя в пула 100 задачи. При добавянето, посочваме метод, който да се изпълни, като използваме делегата **`WaitCallback`**, намиращ се в пространството от имена **`System.Threading`**. Методът **`QueueUserWorkItem(…)`** позволява да подадем към обработката и допълнителен параметър, в случая – името на задачата.

Задачите се изпълняват асинхронно, по реда на постъпването им. В даден момент се изпълняват по няколко задачи, като точният им брой се определя от броя на текущо свободните работни нишки.

Резултатът от изпълнението изглежда така:



Методът ThreadPool.RegisterWaitForSingleObject()

Можем да използваме този метод, когато искаме пула от нишки да чака за някакво събитие. Методът регистрира делегат. Методът, свързан с делегата, се изпълнява както при сигнализирането на това събитие, така и след изтичането на зададен таймаут.

RegisterWaitForSingleObject() – пример

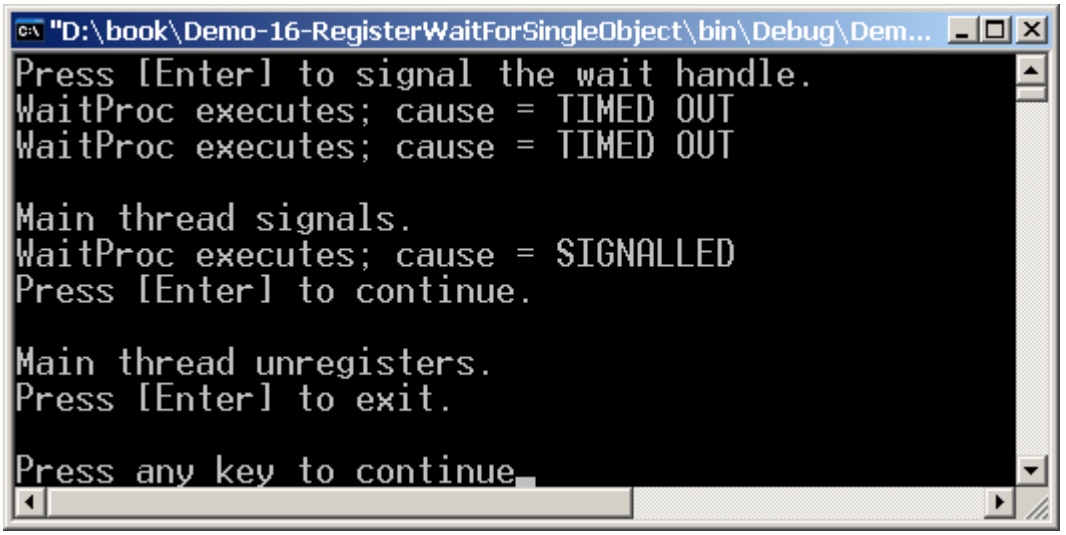
Да разгледаме един пример за използването на метода **ThreadPool.RegisterWaitForSingleObject()**:

```
static void Main()
{
   AutoResetEvent ev = new AutoResetEvent(false);
   object param = "some param";
   RegisteredWaitHandle waitHandle =
      ThreadPool.RegisterWaitForSingleObject(
         ev,  new WaitOrTimerCallback(WaitProc), param, 1000,
           false );
   Console.WriteLine("Press [Enter] to signal the wait handle.");
   Console.ReadLine();

   Console.WriteLine("Main thread signals.");
   ev.Set();
   Console.WriteLine("Press [Enter] to continue.");
   Console.ReadLine();

   Console.WriteLine("Main thread unregisters.");
   waitHandle.Unregister(ev);
   Console.WriteLine("Press [Enter] to exit.");
   Console.ReadLine();
}

public static void WaitProc(object aState, bool aTimedOut)
{
   string cause = aTimedOut ? "TIMED OUT" : "SIGNALLED";
   Console.WriteLine("WaitProc executes; cause = {0}", cause);
}
```

Как работи примерът?

Подобно на метода **QueueUserWorkItem(…)**, **RegisterWaitForSingleObject(…)** създава пула от нишки при своето извикване. Най-напред, посочваме събитието, което чакаме – това е **ev** от тип **AutoResetEvent**. Като използваме делегата **WaitOrTimerCallback**, посочваме метода **WaitProc(…)**, който ще се изпълнява при сигнализиране на събитието. Към метода **WaitProc(…)** можем да подадем произволен параметър – обектът **aState**, на който преди това сме задали стойност "some param". Таймаутът, през който ще се изпълнява метода, е една секунда. Последният параметър определя дали метода да остане регистриран за събитието след първото си изпълнение, дали да се изпълнява на всяка сигнализация на събитието и на всеки изтекъл таймаут. Тъй като стойността му е **false**, методът няма автоматично да бъде дерегистриран след първото си изпълнение.

От този момент нататък, методът започва да се изпълнява на всяка секунда поради изтекъл таймаут. Натискането на [Enter] води до сигнализиране на събитието и еднократно изпълнение на **`WaitProc(…)`**, но вече **`aTimedOut`** има стойност **`false`**. Изпълненията по изтекъл таймаут продължават до достигането на **`waitHandle.Unregister(ev);`**. Дерегистрирането става чрез референцията **`waitHandle`**, върната при регистрирането.



Интерфейсът ISynchronizeInvoke

Когато код, изпълняван в нишката T1, извика метод на обект, този метод обикновено се изпълнява синхронно в същата нишка T1. Понякога обаче се налага изпълнението винаги да протича в нишката, където е създаден обекта (нека я обозначим с T2). Типичен пример за такава необходимост са класовете за форми и контроли в .NET Windows Forms, които трябва винаги да обработват съобщенията в същата нишка, в която са били създадени. За да се справи с подобни случаи, .NET Framework предоставя интерфейса **`System.ComponentModel.ISynchronizeInvoke`**:

```
public interface ISynchronizeInvoke
{
   object Invoke(Delegate method, object[] args);
   IAsyncResult BeginInvoke(Delegate method, object[] args);
   object EndInvoke(IAsyncResult result);
   bool InvokeRequired {get;}
}
```

Използване на ISynchronizeInvoke

`ISynchronizeInvoke` предоставя стандартен механизъм за извикване на методи на обекти, живеещи на други нишки. Нека един обект да

имплементира **ISynchronizeInvoke** и клиентски код на нишка T1 да извика **Invoke(…)** върху този обект. Това ще доведе до следната последователност от действия:

1. Блокиране на извикващата нишка T1.
2. Маршализация на извикването до нишката T2.
3. Изпълнение върху нишката T2.
4. Маршализация на върнатите стойности до нишката T1.
5. Връщане на контрола на нишката T1.

Invoke(…) приема делегат, съответен на метода, който ще бъде изпълнен на T2, и масив от обекти като параметри.

Използване на ISynchronizeInvoke – пример

Ще дадем един пример, в който клас за калкулатор имплементира **ISynchronizeInvoke** и предоставя **Add(…)** метод за събиране на две числа. В кода сме пропуснали същинската реализация на методите на **ISynchronizeInvoke**, а ще концентрираме вниманието си върху начина на ползването на класа в клиентски код. Ето все пак как изглежда скелета на класа **Calculator**.

```
public class Calculator : ISynchronizeInvoke
{
   public int Add(int arg1, int arg2)
   {
      int threadID = Thread.CurrentThread.GetHashCode();
      Console.WriteLine("Callback thread ID is " + threadID);
      return arg1 + arg2;
   }
   // ISynchronizeInvoke implementation here ...
}
```

Ето как се използва класа **Calculator**:

```
public delegate int AddDelegate(int arg1, int arg2);

public void CalculatorInvoke()
{
   int threadID = Thread.CurrentThread.GetHashCode();
   Console.WriteLine("Client thread ID is " + threadID);

   Calculator calc = new Calculator();

   AddDelegate addDelegate = new AddDelegate(calc.Add);
   object[] arr = new object[] {3,4};
   int sum = (int) calc.Invoke(addDelegate,arr);
```

```
   Debug.Assert(sum == 7);
}
```

Един възможен изход, който можем да получим, е следният:

```
Callback thread ID is 29
Client thread ID is 30
```

Как работи примерът?

Тъй като обработката се изпълнява на нишка, различна от тази на клиентския код, можем да извършим асинхронно извикване чрез методите **`BeginInvoke(…)`** и **`EndInvoke(…)`**. Асинхронният механизъм на работа е описан подробно по-надолу в темата.

Свойството **`InvokeRequired`** показва дали клиентската нишка е същата като тази, на която трябва да се изпълни метода на обекта. Ако е същата (т.е. **`InvokeRequired`** е равно на **`false`**), методът може да бъде извикан директно без механизма на **`ISynchronizeInvoke`**.

Windows Forms и ISynchronizeInvoke

Базовите класове в Windows Forms използват **`ISynchronizeInvoke`**. Всеки клас наследник на **`Control`** разчита на Windows съобщения и на опашката от събития, където те биват обработвани в безкраен цикъл. Но съобщенията за даден прозорец се доставят само до нишката, където е бил създаден. Затова, в общия случай, достъпът до Windows Forms класове от друга нишка трябва да става изключително и само през методите на **`ISynchronizeInvoke`**.

Таймери

Често в приложенията, които разработваме, възниква необходимост от изпълняване на задачи през регулярни времеви интервали. Таймерите предоставят такава услуга. Те са обекти, които известяват приложението при изтичане на предварително зададен интервал от време. Таймерите са полезни в редица сценарии, например, когато искаме да обновяваме периодично потребителския интерфейс с актуална информация за статуса на някаква задача или да проверяваме състоянието на променящи се данни.

Такава услуга изглежда на пръв поглед лесна за имплементация. Можем да използваме работна нишка, която заспива за определено време и после известява за събуждането си. Но трябва да реализираме и много други функции: за начало и край на отброяване на времето, за управление на работната нишка, за промяна на интервала, за задаване на функция за обратно извикване.

.NET Framework ни предоставя наготово три различни решения за този проблем. Ще разгледаме кога е удачно да използваме всеки един от класовете, които ще разгледаме.

System.Timers.Timer

Класът **System.Timers.Timer** има следната дефиниция:

```
public class Timer
{
   public Timer();
   public Timer(double interval);

   // Properties
   public bool AutoReset{get; set; }
   public bool Enabled{get; set; }
   public double Interval{get; set;}
   public ISynchronizeInvoke SynchronizingObject { get; set; }

   //Events
   public event ElapsedEventHandler Elapsed;

   // Methods
   public void Close();
   public void Start();
   public void Stop();
   /* Other members */
}
```

Класът предоставя събитие за изтичане на времевия интервал **Elapsed**, което е делегат от тип **ElapsedEventHandler**, дефиниран като:

```
public delegate void ElapsedEventHandler(
   object sender, ElapsedEventArgs e);
```

При изтичане на интервала, указан в свойството **Interval**, таймерът от тип **System.Timers.Timer** ще извика записалите се за събитието методи, използвайки нишка от пула. Ако използваме един и същ метод за получаване на събития от няколко таймера, чрез аргумента **sender** можем да ги разграничим. Класът **ElapsedEventArgs** чрез свойството **DateTime SignalTime** ни предоставя точното време, когато е бил извикван метода.

За стартиране и спиране на известяването, можем да извикаме съответно **Start()** и **Stop()** методите. Свойството **Enabled** ни позволява да инструктираме таймера да игнорира събитието **Elapsed**. Това прави **Enabled** функционално еквивалентно на съответните **Start()** и **Stop()** методи. Когато приключим с таймера, трябва да извикаме **Close()**, за да освободим съответните системни ресурси.

System.Timers.Timer – пример

Ето пример за употребата на **`System.Timers.Timer`**:

```
using System;
using System.Timers;
using System.Threading;

class SystemTimerClient
{
   System.Timers.Timer mTimer;
   int mCounter = 0;

   public SystemTimerClient()
   {
      mTimer = new System.Timers.Timer();
      mTimer.Interval = 1000; // One second
      mTimer.Elapsed += new ElapsedEventHandler(OnTick);
      mTimer.Start();

      //Can block, because the Timer uses thread from thread pool
      Thread.Sleep(4000);

      mTimer.Stop();
      mTimer.Close();
   }

   private void OnTick(object source, ElapsedEventArgs e)
   {
      string tickTime = e.SignalTime.ToLongTimeString();
      mCounter++;
      Console.WriteLine(mCounter.ToString() + " " + tickTime);
   }

   private static void Main()
   {
      SystemTimerClient obj = new SystemTimerClient();
   }
}
```

Резултатът от изпълнението на програмата е:

```
1 16:13:31
2 16:13:32
3 16:13:33
```

Как работи примерът?

Тъй като методът, който е обработчик на събитието за изтичане на интервал, се изпълнява в отделна нишка, трябва да осигурим синхронизиран достъп до член-променливите на обекта.

Свойството **SynchronizingObject** ни позволява да укажем обект, имплементиращ **ISynchronizeInvoke**. Той ще бъде използван от таймера за изпълнението на функцията за обратно извикване в определена нишка, вместо в нишка, принадлежаща на пула. Това е удобно, примерно, когато имаме таймер от тип **System.Timers.Timer** в клас, наследник на **Windows.Forms.Form**. Ако укажем самата форма на свойството **SynchronizingObject**, то методът обработчик на **Elapsed** ще се изпълни в основната нишка на потребителския интерфейс, където безопасно можем да променяме свойствата на формата и контролите й.

Visual Studio .NET има вградена поддръжка за **System.Timers.Timer** в дизайнера си. Можем директно да привлачим такъв обект от раздела компоненти върху Windows форма, ASP.NET форма или уеб услуга и да му укажем съответните свойства. В случая на Windows Forms, дизайнерът на VS.NET автоматично указва свойството **SynchronizingObject** на инстанцията на самата форма.

System.Threading.Timer

Пространството от имена **System.Threading** съдържа друг клас за таймер, който е със следната дефиниция:

```
public sealed class Timer : MarshalByRefObject, IDisposable
{
   public Timer(TimerCallback callback,
      object state, long dueTime, long period);

   /* More overloaded constructors */

   public bool Change(int dueTime, int period);

   /* More overloaded Change() */

   public virtual void Dispose();
}
```

System.Threading.Timer прилича на **System.Timers.Timer** и също използва пула с нишки. Основната разлика е, че той позволява малко по-разширен контрол – може да указваме кога таймера да започне да отброява, както и да предаваме всякаква информация на метода за обратни извиквания чрез обект от произволен тип. За да ползваме **System.Threading.Timer**, трябва в конструктора му да подадем делегат от тип **TimerCallback**, дефиниран като:

```
public delegate void TimerCallback(object state);
```

При всяко изтичане на времевия интервал, ще бъдат извиквани методите в този делегат. Обикновено като обект за състояние има полза да подаваме създателя на таймера, за да можем да използваме същия метод

за обратни извиквания за обработка на събития от множество таймери. Другият параметър в конструктора на таймера е времевият интервал. Той може и да бъде променен впоследствие с извикване на **Change(…)** метода.

System.Threading.Timer не предлага удобен начин за стартиране и спиране. Неговата работа започва веднага след конструирането му (точно след изтичането на подаденото стартово време) и прекъсването му става само чрез **Dispose()**. Ако искаме да го рестартираме трябва да създадем нов обект.

System.Threading.Timer – пример

Ето един пример за употребата на **System.Threading. Timer**:

```
using System;
using System.Threading;

class ThreadingTimerClient
{
   private Timer mTimer;
   private int mCounter = 0;

   public ThreadingTimerClient()
   {
      Start();
      Thread.Sleep(4000);
      Stop();
   }

   private void Start()
   {
      TimerCallback callBack = new TimerCallback(OnTick);
      mTimer = new Timer(callBack, null, 0, 1000);
   }

   private void Stop()
   {
      mTimer.Dispose();
      mTimer = null;
   }

   private void OnTick(object state)
   {
      mCounter++;
      Console.WriteLine(mCounter.ToString());
   }

   private static void Main()
   {
      ThreadingTimerClient obj = new ThreadingTimerClient();
   }
```

```
}
```

Резултатът от изпълнението на програмата е:

```
1
2
3
4
```

System.Windows.Forms.Timer

Пространството от имена **System.Windows.Forms** съдържа още един клас за таймер, който е със следната дефиниция:

```
public class Timer : Component, IComponent, Idisposable
{
   public Timer();

   public bool Enabled{virtual get ;virtual set;}
   public int Interval {get; set;}

   public event EventHandler Tick;

   public void Start();
   public void Stop();
}
```

Въпреки, че методите на **System.Windows.Forms.Timer** много приличат на тези на **System.Timers.Timer**, то **System.Windows.Forms.Timer** не използва пула с нишки за обратните извиквания към Windows Forms приложението. Вместо това, през определено време той пуска Windows съобщението **WM_TIMER** в опашката за съобщения на текущата нишка.

Използването на **System.Windows.Forms.Timer** се различава от употребата на **System.Timers.Timer**, само по сигнатурата на делегата за обратни извиквания, който в случая е стандартният **EventHandler**.

VS.NET има вградена поддръжка за **System.Windows.Forms.Timer** в дизайнера си. Можем директно да привлачим такъв обект от раздела Windows Forms върху Windows форма.

Тъй като при Windows Forms таймерите всички функции за обратни извиквания се изпълняват на главната нишка за потребителския интерфейс, то няма нужда от допълнителна синхронизация. Това обаче може да е проблем, защото при времеотнемащи операции приложението няма да може да отговаря бързо.

Как да изберем таймер?

Ако разработваме Windows Forms приложение, обикновено е най-лесно да използваме **System.Windows.Forms.Timer**. В повечето други случаи е по-удачно да ползваме **System.Timers.Timer**. Методите му изглеждат по-интуитивни и по-удобни от тези на **System.Threading.Timer**.

Volatile полета

Ако кодът ни използва публични полета, то оптимизациите, които извършва компилаторът, могат да доведат до неочаквани проблеми. Ако стойността на такава променлива се прочита няколко пъти, компилаторът може да я кешира при първото четене във временна локална променлива, вместо да осъществява достъп до нея през обекта, на когото принадлежи. Да разгледаме следния пример:

```
class MyClass
{
   public int Number;

   public static void Main()
   {
      MyClass obj = new MyClass();
      int num1 = obj.Number;
      int num2 = obj.Number; //Compiler may use cached value here
   }
}
```

Оптимизациите при компилация могат да доведат до подобрена производителност, особено в цикли. Проблемът е, че ако настъпи превключване на активната нишка след инициализацията на **num1** и преди тази на **num2**, и друга нишка промени стойността на **Number**, то **num2** ще съдържа старата кеширана стойност.

Ако искаме да използваме такива публични полета (а по-препоръчително е използването на свойства) без да синхронизираме изрично достъпа до тях, можем да се възползваме от **volatile** полетата, които се поддържат от компилатора на C#. Те се дефинират с ключовата дума **volatile**:

```
public volatile int Number;
```

При **volatile** полета, компилаторът не кешира стойността им, а винаги я прочита наново. Във Visual Basic.NET няма еквивалент на C# ключовата дума **volatile**. Препоръчваме вместо да се ползват **volatile** полета, да си заключваме изрично обекта или полетата, за да гарантираме безопасен достъп до тях.

Асинхронни извиквания

Асинхронните извиквания са мощен механизъм за паралелно изпълнение на няколко задачи, при който не е необходимо изрично да се създава нова нишка за всяка задача.

Какво е асинхронно извикване?

По подразбиране методите в кода на програмата се изпълняват синхронно, тоест изпълнението преминава на следващия оператор чак след като приключи текущият метод. При асинхронното извикване не се изчаква края на изпълнението на текущия оператор, а веднага се преминава на следващия. Обработката на асинхронното извикване се извършва в отделна нишка, която обикновено е от стандартния пул с нишки.

Къде се ползва асинхронно извикване?

В .NET Framework широко се използват асинхронни извиквания при вход-изход от файлови и други потоци, при мрежови операции с HTTP и TCP, при отдалечено извикване с Remoting, при ASP.NET XML уеб услуги и други. Асинхронното програмиране се реализира лесно в нашия код с помощта на делегати. Като алтернатива можем и сами да предоставим явен асинхронен интерфейс за нашите класове, както ще видим малко по-късно.

Асинхронно извикване чрез делегат

Делегатите предоставят възможност за лесно асинхронно извикване на синхронни методи. Трябва само да създадем делегат със сигнатура, съответна на метода и можем да използваме функциите за започване на асинхронно извикване: **`BeginInvoke(…)`** и за изчакване на получаване на резултата: **`EndInvoke(…)`**.

Асинхронно извикване чрез делегат – пример

Можем да илюстрираме казаното с прост пример, описващ асинхронно сумиране на две цели числа:

```
using System;
using System.Threading;

class AsyncCallDemo
{
  public delegate int SumDelegate(int a, int b);

  public int Sum(int a, int b)
  {
    Thread.Sleep(3000);
    return a + b;
```

```
    }

    static void Main()
    {
        SumDelegate asyncCall = new SumDelegate(
            new AsyncCallDemo().Sum);

        Console.WriteLine("Starting method async.");
        IAsyncResult status = asyncCall.BeginInvoke(5, 6, null,
            null);
        Console.WriteLine("Async method is now working...");

        Console.WriteLine("Calling EndInvoke()...");
        Console.WriteLine("It will block until method finishes.");
        int result = asyncCall.EndInvoke(status);
        Console.WriteLine("EndInvoke() returned.");
        Console.WriteLine("Result = {0}", result);
    }
}
```

Като резултат от изпълнението, ще получим следния изход:

```
Starting method async.
Async method is now working...
Calling EndInvoke()...
It will block until method is finished.
EndInvoke() returned.
Result = 11
```

В метода **`Sum(…)`** сме сложили реда **`Thread.Sleep(3000)`** и затова след съобщението "It will block until method is finished." се получава близо 3-секундно забавяне. Извикването на **`EndInvoke(…)`** блокира изпълнението на текуща нишка, докато не приключи съответното асинхронно извикване.

Модел за асинхронно програмиране

Ползването на делегати за асинхронно извикване е удобно, защото не изисква писане на много код. Има обаче случаи, в които се налага изрично да имплементираме асинхронно извикване на метод. Това е необходимо, когато бързодействието е критично (използването на делегати може да е тежко) или ако методът трябва да се извиква само асинхронно. В такива случаи се препоръчва следването на утвърдения в .NET Framework модел за асинхронни извиквания, с който ще се запознаем сега.

Сигнатура на методите за асинхронни извиквания

Нека да предоставим асинхронната версия на функцията **`Sum(…)`**, която сумира две целочислени числа. В .NET Framework методите предназначени за асинхронно извикване използват нотацията **`BeginXXXXX(…)`** и

EndXXXXX(…), където **XXXXX** е синхронната версия на метода. В случая трябва да дефинираме **BeginSum(…)** и **EndSum(…)**, за да направим стандартна асинхронна версия на метода **Sum(…)**:

```
IAsyncResult BeginSum(int a, int b,
  AsyncCallback requestCallback,
  object stateObject
```

AsyncCallback е делегат към метод, който да се извика след приключване изпълнението на асинхронното извикване. Ако подадем **null**, няма да се изпълни нищо след завършването.

```
int EndSum(IAsyncResult ar);
```

На блокиращия метод **EndSum(…)** му се подава **IAsyncResult**, върнат като резултат от **BeginSum(…)** и така се изчаква приключването на работата на асинхронния метод.

Интерфейсът IAsyncResult

Ето какви свойства предоставя интерфейсът **IAsyncResult**:

```
interface IAsyncResult
{
  object AsyncState {get;}
  WaitHandle AsyncWaitHandle {get;}
  bool CompletedSynchronously {get;}
  bool IsCompleted {get;}
}
```

AsyncState връща същия обект, подаден като **stateObject** на **BeginSum()**. Това е начин за следене на статуса на работа и само асинхронно извикваният метод трябва да го променя.

AsyncWaitHandle се използва като параметър на методите **WaitAll(…)**, **WaitOne()** или **WaitAny(…)** на класа **WaitHandle** за изчакване приключването на асинхронния метод.

CompletedSynchronously връща **true**, ако асинхронният метод е приключил бързо работа, още преди края на извикването на **BeginXXXXX(…)**.

IsCompleted връща **true** ако асинхронният метод е приключил своята работа. Чрез механизма "polling" можем през определено време да проверяваме истинността на **IsCompleted**, докато върне **true**.

Проверка за приключване на асинхронното извикване

Има четири начина да проверим дали е приключил един асинхронен метод

- Чрез механизма "polling" проверяваме **`IAsyncResult.IsCompleted`** през определено време.
- Чрез някои от методите за синхронизация на **`WaitHandle`** с параметър свойството **`IAsyncResult.AsyncWaitHandle`**. Можем и да зададем таймаут, за да не се чака безкрайно дълго.
- Чрез извикване на **`EndXXXXX(…)`**, който блокира изпълнението, докато асинхронният метод не свърши работата си.
- Чрез подаване на метод за обратно извикване на **`BeginXXXXX(…)`** през делегата **`AsyncCallback`**, който приема единствен параметър от тип **`IAsyncResult`**. Подаденият метод ще бъде извикван, когато асинхронният метод приключи работа. Имаме достъп до резултата чрез свойството **`AsyncState`**.

Изчакване на асинхронна операция – няколко примера

Ще демонстрираме изброените подходи с един пример, в който асинхронно четем данни от файл. Първо ще разгледаме някои общи променливи и методи на класа **`FileReaderDemo`**, а после поотделно функциите, реализиращи всеки един от подходите:

```
using System;
using System.IO;
using System.Text;
using System.Threading;

internal class FileReaderDemo
{
   private const string FILE_NAME = "data.txt";
   private const int READ_BUF_SIZE = 8192;
   private const int WAIT_TIMEOUT = 50;

   private Stream GetFileStream(string aFileName)
   {
      FileStream stream =
         new FileStream(
         aFileName, FileMode.Open,
         FileAccess.Read, FileShare.Read,
         READ_BUF_SIZE, true);
      return stream;

   }

    ...
}
```

Асинхронно четене с polling:

```
public void AsynchronousPollReadFile()
{
```

```
   Stream stream = GetFileStream(FILE_NAME);
   byte[] buf = new byte[READ_BUF_SIZE];
   IAsyncResult readResult = stream.BeginRead(
     buf, 0, buf.Length, null, null);

   Console.Write("Asynchronous Poll Read");
   while (!readResult.IsCompleted)
   {
     Thread.Sleep(WAIT_TIMEOUT);
     Console.Write(".");
   }
   Console.WriteLine();

   using (stream)
   {
     int bytesRead = stream.EndRead(readResult);
     string data = Encoding.ASCII.GetString(buf, 0, bytesRead);
     Console.WriteLine("\tCount of bytes: {0}", bytesRead);
     Console.WriteLine("\tData: {0}\n", data);
   }
}
```

Асинхронно четене с **`WaitHandle`**:

```
public void AsynchronousWaitReadFile()
{
   Stream stream = GetFileStream(FILE_NAME);
   byte[] buf = new byte[READ_BUF_SIZE];
   IAsyncResult readResult = stream.BeginRead(
     buf, 0, buf.Length, null, null);

   Console.Write("Asynchronous Wait Read");
   bool finished;
   do
   {
     finished = readResult.AsyncWaitHandle.
       WaitOne(WAIT_TIMEOUT, false);
     Console.Write(".");
   } while (! finished);
   Console.WriteLine();

   using (stream)
   {
     int bytesRead = stream.EndRead(readResult);
     string data = Encoding.ASCII.GetString(buf, 0, bytesRead);
     Console.WriteLine("\tCount of bytes: {0}", bytesRead);
     Console.WriteLine("\tData: {0}\n", data);
   }
```

Асинхронно четене с **`EndRead(…)`**:

```
public void AsynchronousEndReadFile()
{
   Stream stream = GetFileStream(FILE_NAME);
   using (stream)
   {
      byte[] buf = new byte[READ_BUF_SIZE];
      IAsyncResult readResult = stream.BeginRead(
         buf, 0, buf.Length, null, null);
      int bytesRead = stream.EndRead(readResult);
      string data = Encoding.ASCII.GetString(buf, 0, bytesRead);
      Console.WriteLine("Asynchronous End Read");
      Console.WriteLine("\tCount of bytes: {0}", bytesRead);
      Console.WriteLine("\tData: {0}\n", data);
   }
}
```

Да завършим с пример за асинхронно четене с метод за обратно извикване. Нужен ни е един помощен клас **`FileReadState`** за състоянието на операцията. Впоследствие в метода **`OnReadCompleted(…)`** ще го използваме за обработка на крайния резултат:

```
public void AsynchronousCallbackReadFile()
{
   Stream stream = GetFileStream(FILE_NAME);
   byte[] buf = new byte[READ_BUF_SIZE];
   FileReadState state = new FileReadState(stream, buf);
   AsyncCallback readDone = new
      AsyncCallback(this.OnReadCompleted);
   IAsyncResult readResult = stream.BeginRead(
      buf, 0, buf.Length, readDone, state);
}

private void OnReadCompleted(IAsyncResult aResult)
{
   FileReadState state = (FileReadState) aResult.AsyncState;
   Stream stream = state.Stream;
   using (stream)
   {
      int bytesRead = stream.EndRead(aResult);
      byte[] buf = state.Buffer;
      string data = Encoding.ASCII.GetString(buf, 0, bytesRead);
      Console.WriteLine("Asynchronous Callback Read");
      Console.WriteLine("\tCount of bytes: {0}", bytesRead);
      Console.WriteLine("\tData: {0}\n", data);
   }
}

internal class FileReadState
{
```

```
   private Stream mStream;
   private byte[] mBuffer;

   public Stream Stream
   { get { return mStream; } }

   public byte[] Buffer
   { get { return mBuffer; } }

   public FileReadState(Stream aStream, byte[] aBuffer)
   {
      mStream = aStream;
      mBuffer = aBuffer;
   }
}
```

Упражнения

1. Напишете програма, която стартира предварително зададен брой нишки. Всяка нишка изписва "Thread X started", спи (Thread.Sleep()) случаен брой милисекунди и изписва "Thread X stopped". X трябва да се задава в конструктора на класа, който съдържа метода, използван в ThreadStart делегата.
2. Напишете Windows Forms приложение, което да търси зададен текст във всички файлове от указана директория (подобно на търсенето от Windows Explorer) като използвате нишки. Реализирайте по правилен начин прекратяване на търсенето.
3. Разгледайте решението на проблема "производител/консуматор". Направете примерно приложение, с което да тествате дали предложената реализация работи коректно.
4. Решете проблема "обядващи философи" чрез подходящи синхронизационни механизми. Направете приложение, с което да тествате дали работи правилно.

Използвана литература

1. Михаил Стойнов, Многонишково програмиране и синхронизация, http://www.nakov.com/dotnet/lectures/Lecture-16-Concurrency-v1.0.ppt
2. Juval Lowy, "Programming .NET Components", O'Reilly, 2003, ISBN 0596003471
3. Tom Archer, Andrew Whitechapel, "Inside C# 2nd Edition", Microsoft Press, 2002, ISBN 0735616485
4. MSDN Library – http://msdn.microsoft.com

Глава 18. Мрежово и Интернет програмиране

Автори

Георги Пенчев

Ивайло Христов

Необходими знания

- Базови познания за .NET Framework
- Базови познания за езика C#
- Базови познания по компютърни мрежи, TCP/IP, протоколи и услуги
- Познания по многонишково програмиране и синхронизация

Съдържание

- OSI модел. Основни мрежови понятия
- IP адрес, DNS, порт, мрежов интерфейс
- TCP, UDP, сокет връзки
- Основни мрежови услуги
- Класове за мрежово програмиране
- Комуникация по TCP – **`TcpClient`**, **`TcpListener`**
- Обслужване на много клиенти едновременно
- Комуникация по UDP – **`UdpClient`**
- Класовете **`IpAddress`**, **`Dns`**, **`IPEndPoint`**
- Сокети на ниско ниво с класа **`Socket`**
- Достъп до Интернет ресурси по URI – **`WebClient`**, **`HttpWebRequest`**, **`HttpWebResponse`**
- Протоколи за работа с e-mail. Изпращане и получаване на e-mail
- Класове за изпращане на e-mail. Прикрепени файлове (attachments)

В тази тема ...

В настоящата тема ще разгледаме някои основни средства, предлагани от .NET Framework за мрежово програмиране. Ще започнем с кратко въведение в принципите на работа на съвременните компютърни мрежи и Интернет и ще разгледаме протоколите, чрез които се осъществява мре-

жовата комуникация. Обект на дискусия в темата са както класовете за програмиране на ниво TCP и UDP сокети, така и някои класове, предлагащи по-специфични възможности – представяне на IP адреси, изпълнение на DNS заявки и др. В края на темата ще се спрем на средствата за извличане на уеб-ресурси от Интернет, както и на класовете за работа с e-mail в .NET.

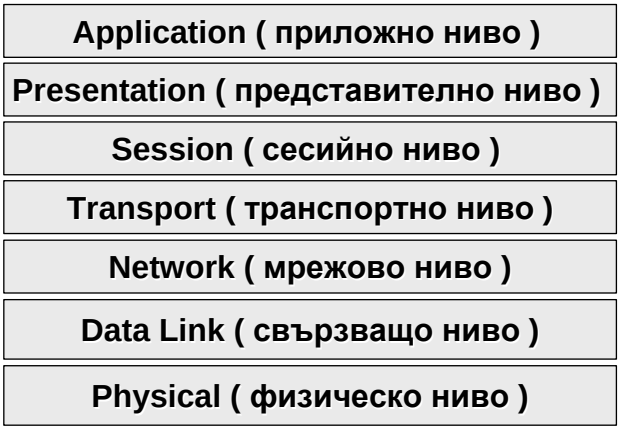
OSI модел

За намаляване на сложността мрежите са разделени на слоеве. Всеки слой използва услугите на слоя, намиращ се под него, без да се интересува от това как работи той или по-горният слой. Ето една житейска ситуация, в която задълженията също са разделени на слоеве.

Нека разгледаме голямата софтуерна компания "Марс". Наближава новогодишното празненство на фирмата и шефът казва на секретарката да изпрати покана на предания клиент г-н Христов. Шефът не се интересува дали секретарката ще изпрати електронна поща, дали ще се обади по телефона или ще използва услугите на местната пощенска служба. За него е достатъчно само да нареди да се изпрати поканата. Секретарката от своя страна решава, че най-удачно е да изпрати писмото чрез пощенската служба – написва адреса, залепя марки и пуска писмото. Секретарката също не се интересува от това дали клиентът живее в съседния квартал или в някой далечен град. Това е работа на пощенската служба, която ще се погрижи за доставката на поканата използвайки кола, влак или в някои случаи самолет.

Ето как всеки слой се грижи за собствените си задължения и не се интересува от детайлите от работата на останалите слоеве.

Според световно възприетите стандарти за компютърни мрежи на организацията IEEE (Institute of Electrical and Electronics Engineers) комуникациите във всяка мрежа се осъществяват на следните 7 нива:



Комуникацията на всяко ниво зависи от специален набор инструкции, наречен протокол, който указва как трябва да се интерпретира информацията, получена от отсрещната страна.

Физическо ниво

Физическото ниво се грижи за пренасянето на данни през комуникационната среда. Основна функция на този слой е да управлява кодирането и декодирането на сигналите, представляващи двоичните цифри 0 и 1.

Свързващо ниво (канално ниво)

Грижи се за обмена на блокове данни между двете системи, като открива и евентуално коригира възникналите грешки. Също така управлява достъпа до комуникационната среда на базата на някой от протоколите: Ethernet, Token ring, PPP и др.

Мрежово ниво

Осигурява маршрутизацията на единици информация от машината–източник до машината–получател. Типични протоколи са: IPv4, IPv6, ICMP, IGMP, X.25, IPX и др.

Транспортно ниво

Това ниво позволява както пренасянето на отделни пакети, така и създаването на надеждни комуникационни канали за пренос на данни. Грижи се за създаване, поддръжка и затваряне на комуникационните канали. Някои от протоколите за транспортното ниво са: TCP, UDP, RTP, SPX и др.

Сесийно ниво

Организира и синхронизира прозрачната обмяна на информация между два процеса в операционните системи на комуникационните машини. Типични протоколи са: RPC, NetBIOS, X.225 и др.

Представително ниво

Представителното ниво осигурява общ формат за представяне на данните по време на техния обмен в мрежата. Това се налага, за да бъде възможно комуникирането между компютри с различно представяне на данните. Типични схеми за унифициране на данните са : XDR, ASN.1, SMB и др.

Приложно ниво

Протоколите от това ниво задават форматите и правилата за обмен на данни между комуникиращите приложения. Такива протоколи са например: HTTP, SMTP, POP3, DNS и др.

Основи на мрежовото програмиране

В тази част ще се запознаем с някои основни понятия и протоколи за мрежово програмиране.

IP адрес

IP адресът е уникален адрес, използван от мрежови устройства (обикновено компютри). Използва се за разпознаване на устройствата, когато те си комуникират. IP адресът е като телефонен номер – когато искате да се свържете с определен телефонен пост, използвате неговия номер.

Сегашният стандартен протокол в мрежите е IP версия 4. Неговите адреси са с големина 32 бита. Обикновено се записват като четири осембитови числа, разделени с точка, например: 194.145.63.12 или 212.50.1.217. Тази версия на протокола може да предостави над 4 милиарда различни адреса. В последните години се оказва, че това не е достатъчно.

Разработен е нов протокол – IP версия 6, който все още не е широко разпространен. Адресите от този протокол са с големина 128 бита. Техният брой е 2^{128}, което е приблизително равно на 3.403×10^{38}. Обикновено адресите се записват като осем шестнайсетични числа в интервала 0-FFFF, например: 2001:0db8:85a3:08d3:1319:8a2e:0370:7334.

Domain Name Service (DNS)

Компютрите в Интернет се разпознават чрез IP адреси, но тези числови идентификатори не са лесни за запомняне от човек. Повечето хора предпочитат да работят с имена. Ако искате да прочетете новините от страницата на вестник "Капитал", ще ви е по-лесно да се сетите за www.capital.bg, вместо за адреса 193.194.140.15. Затова е създадена системата DNS (Domain Name Service), която служи за управление на съответствията между IP адреси и имена (наричани домейни). Тя може да преобразува имена в адреси и обратно.

Порт

На един и същ компютър обикновено работят повече от едно приложения. В общия случай компютърът има само една физическа връзка към мрежата. Тази връзка може да бъде използвана за комуникация с повече от едно приложение. Как различаваме кои данни за кое приложение са? Използвайки 16-битово число, наричано порт, разграничаваме комуникационните канали на различните приложения един от друг. Изпращачът, изпращайки данни за даден компютър, подава и номер на порт. Така информацията достига до правилния си получател – конкретното приложение, отговарящо на този порт. Често се казва, че дадено приложение "слуша" на определен порт.

Основни мрежови услуги

В мрежата работят много стандартни услуги. Прието е някои от популярните мрежови приложения да имат стандартен порт по подразбиране. Затова номерата на портове до 1024 са запазени за стандартни услуги. За да се избегнат проблеми със засичането на две приложения, които се опитват да използват един и същ порт, се препоръчва, ако разработвате приложение, което "слуша" на даден порт, да изберете порт над 1024. Подробен списък с портовете, използвани от популярни приложения, може да намерите на http://www.iana.org/assignments/port-numbers.

Ето някои от най-известните мрежови услуги:

Услуга	Порт	Описание
HTTP	80	Достъп до уеб сайтове, ресурси и услуги
SMTP	25	Изпращане на e-mail
POP3	110	Извличане на e-mail
FTP	21	Достъп до отдалечени файлове
DNS	53	Извличане на IP по име на сървър и обратното
SSH	22	Сигурен достъп до отдалечен терминал

Мрежов интерфейс

Мрежовият интерфейс е абстрактна структура, чрез която операционната система управлява изпращането и приемането на информация по мрежата. Възможно е една машина да бъде свързана към няколко мрежи едновременно. Тогава към всяка мрежа машината има различен мрежов интерфейс. Всеки интерфейс има различен IP адрес, съответно и машината има повече от един IP адрес.

Loopback интерфейс

Повечето имплементации на IP протокола предоставят служебен интерфейс за обратна връзка към локалната машина. Целият трафик, изпратен през този интерфейс, се връща обратно на компютъра, който го е изпратил. Най-използваният IP адрес на Loopback интерфейса е `127.0.0.1` за IP версия 4. Стандартният домейн, отговарящ на този интерфейс, е **`localhost`**. Това е много полезно за програмистите, разработващи мрежови приложения, защото позволява разработването и тестването на мрежови приложения, без да е нужно компютърът, на който се разработва приложението, да е свързан към мрежа.

Протоколът TCP

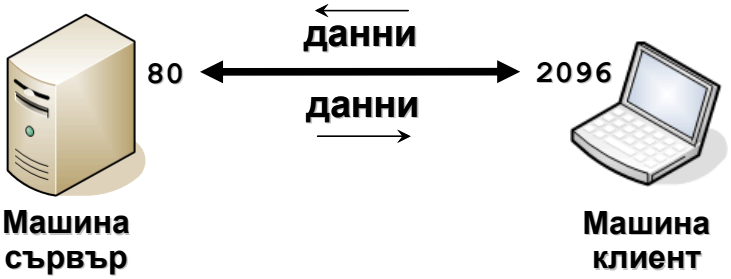
Протоколът TCP е един от най-широко разпространените протоколи за мрежова комуникация. Този протокол създава надежден двупосочен комуникационен канал за обмен на данни. Това гарантира, че изпратените данни ще пристигнат в същия ред, в който са изпратени. Ако данните не могат да се изпратят или получат, ще възникне грешка. Комуникационният канал съществува, докато някоя от двете страни не го прекрати. Комуникацията по протокола TCP се използва в приложения, в които редът на пристигане на данните и надеждността са важни.

Протоколът UDP

Протоколът UDP позволява изпращане и приемане на малки независими един от друг пакети с данни, наречени **datagram пакети**. Не гарантира реда на пристигане на datagram пакетите, нито че те изобщо ще пристигнат. За сметка на това е по-бърз от протокола TCP. Използва се в приложения, в които скоростта на предаване на данните е по-важна от надеждността. Например, ако гледате видео материал по Интернет, няма да е от голямо значение, ако от време на време вместо една точка от екрана се появи звездичка. Но ще е от голямо значение, ако кадрите се забавят и се получава завличане на образа. В такива приложения е логично да се използва протоколът UDP.

Как две отдалечени машини си "говорят"?

Ако искаме да осъществим връзка между два компютъра и да разменим определени данни, се нуждаем от приложение "клиент" и приложение "сървър":



Първо трябва да стартираме сървърното приложение, като го накараме да "слуша" на даден порт. Нека това е порт 80. Клиентското приложение се стартира на компютъра–клиент и се опитва да установи комуникационен канал, свързвайки се със сървърния компютър, като указва IP адреса и порта, към които иска да се свърже. За да е успешна комуникацията, е нужно клиентът да може да изпраща данни на сървъра, но и сървърът да може да изпраща на клиента. Когато сървърът изпраща данни на клиента, се нуждае не само от IP адрес, но и от порт. За целта или клиентът сам определя порта, или операционната система му задава такъв. След като

комуникационният канал е създаден, успешно могат да се обменят данни, докато една от двете страни не прекрати връзката.

Класове за мрежово програмиране в .NET

Мрежовото програмиране на практика се състои в писане на код, който да управлява обмена на пакети данни по мрежата и да обработва получената информация. Класовете, които .NET Framework предлага за това, са разпределени в две основни именни пространства – **System.Net** и **System.Net.Sockets**. Чрез опростени класове като **TcpClient**, **TcpListener** и **UdpClient** лесно можем да реализираме комуникация съответно по TCP и UDP протокол. Освен тях, можем да използваме по-функционалния клас **Socket**, както и множеството помощни класове за програмиране на приложно ниво (application layer), чрез които да реализираме и да използваме съответните услуги (уеб-програмиране, DNS услуги, пощенски услуги и т.н.).

Пространството System.Net.Sockets

Тук се намират споменатите по-горе основни класове за осъществяване на комуникация чрез сокети, както и няколко помощни класа, на които няма да се спираме подробно – класове за опции, за изключения и за мрежово програмиране при мобилни устройства.

Класовете **TcpClient** и **TcpListener** служат за реализиране на връзка по TCP протокола. Първият клас се използва в клиентската част от приложението и чрез него се свързваме по TCP с отворен порт на отдалечена машина. Методите му позволяват връзка с определен сокет и приемане и изпращане на данни. **TcpListener** реализира сървърната част на връзката – чрез него "слушаме" на определен порт за идващи заявки връзки и установяваме връзка със съответния сокет.

Класът **UdpClient** изпълнява задачата за осъществяване на комуникация по UDP протокола. Както обяснихме, тази комуникация не включва установяване и поддържане на комуникационен канал, както е при TCP, ето защо този клас е достатъчен за реализирането ѝ.

Класът **Socket** реализира абстракцията на Berkeley Sockets API (http://www.answers.com/topic/berkeley-sockets) и е значително по-функционално обобщение на предните три класа. Чрез него можем да осъществим връзка по който и да е от протоколите на мрежово и по-ниски нива от OSI модела, например IP, IPv6, ICMP, IDP и други. Класът има методи както за слушане за връзки и установяване на връзка (connection), така и за изпращане и получаване на данни. Чрез класа **Socket** можем също да осъществяваме и асинхронно предаване на данни.

Последният по-важен клас от това пространство е класът **NetworkStream** – специализация на обикновения клас за поток, който реализира специфичните за мрежов трансфер на данни особености.

Останалите членове на пространството **System.Net.Sockets** са различни изброени типове и помощни класове като например **ProtocolType**, **SocketOptionLevel**, **MulticastOption** и др. Ще ги разгледаме по-подробно в контекста на използващите ги класове. Пространството съдържа и класа за изключения **SocketException**.

Пространството System.Net

Това пространство съдържа по-общ набор от класове, някои от които реализират услуги от приложно ниво, други са помощни класове, които използваме за удобство, трети служат за опции и т.н.

Чрез класовете **HttpWebRequest** и **HttpWebResponse** можем да използваме HTTP услугите и да осъществяваме заявки с този протокол до различни уеб-ресурси. Чрез обработката на тези заявки и отговорите им можем лесно да построим прост вариант на обикновен уеб-браузър.

Класът **Dns** и методите му ни дават достъп до DNS услугите за извличане на име на машина по IP адреса й в мрежата и обратното.

Класовете **Cookie**, **CookieCollection** и **CookieContainer** служат за обработка на бисквитки (cookies). Това са малки текстови файлове, които се намират на клиентските машини. Когато потребител се автентикира в някой сайт, в такива файлове се запазва различна информация като лични настройки и т.н., която после може да се чете от сървърното приложение.

Класовете **IPAddress**, **IPHostEntry** и **IPEndPoint** служат за съхраняване на IP адреси. Първият представя един IP адрес, вторият е списък от съответни адреси и имена (по DNS), а третият е двойка от адрес и номер на порт.

WebRequest и **WebResponse** са абстрактни класове, които съдържат общите операции, необходими за осъществяване на заявки към мрежови ресурси. Една тяхна имплементация са **HttpWebRequest** и **HttpWebResponse**, които разгледахме, а друга двойка наследници са **FileWebRequest** и **FileWebResponse**, които осъществяват достъп до ресурси във файловата система чрез URI със схема **file://xxxx**.

Класът **WebClient** е обобщен клас, чрез който можем да осъществяваме достъп до произволен ресурс чрез **URI (Uniform Resource Identificator)** във файловата система, интернет или локална мрежа.

Чрез обектите от тип **NetworkCredential** можем да пазим информация за потребителско име и парола при автентикация към различни адреси в мрежата, а класовете **SocketPermission**, **DnsPermission** и **WebPermission** контролират достъпа до съответните ресурси. Тях няма да разглеждаме подробно.

Останалите класове, делегати и изброени типове в пространството са с по-ограничена употреба и служат за помощни на основните.

Представяне на IP адреси в .NET Framework

Преди да пристъпим към мрежовото програмиране, добре е да познаваме начините, по които адресите на машините в мрежата се представят в .NET Framework. Макар че методите, които ще използваме, често позволяват и директно изписване на адреса като низ или число, платформата предлага няколко класа, които капсулират абстракцията на мрежов адрес, и са удобни за използване.

Класът IPAddress

Както вече споменахме, всеки обект от този клас представя точно един IP адрес (обикновени IPv4 адреси, както и IPv6 адреси). Можем да създаваме **`IPAddress`** обекти чрез конструкторите им или чрез статичния метод **`Parse(string)`**.

Конструкторът **`IPAddress(byte[])`** приема за аргумент масив от байтове, които отговарят на байтовете на IP адреса, който искаме да представим. Обърнете внимание, че поради бъг в .NET Framework 1.1 този конструктор работи коректно само с IPv6 адреси (приема само 16-байтови масиви).

Конструкторът **`IPAddress(long)`** инициализира IP адреса посредством цяло положително число, което се получава като последователност от байтовете, които съставят адреса.

Методът **`Parse(string)`** се използва най-често за създаване на обекти от тип **`IPAddress`**. Той получава един параметър, който представлява стандартния вид за записване на IP адреси в мрежата – "D.D.D.D" (четири десетични числа в интервала 0-255) за IPv4 и "H:H:H:H:H:H:H:H" (осем шестнайсетични числа в интервала 0-FFFF) за IPv6. Такъв е и видът на резултата от прилагането на метода **`ToString()`** върху обект от класа **`IPAddress`**.

Забележете, че за използването на методи като **`Parse(string)`** и **`ToString()`** с адреси тип IPv6, този протокол трябва да е инсталиран на операционната система. При Windows 2003 Server това е така по подразбиране, но за Windows 2K и Windows XP трябва да се инсталира ръчно, инструкции за което могат да се намерят на сайта на Microsoft (http://www.microsoft.com/technet/prodtechnol/winxppro/plan/faqipv6.mspx).

Следният пример показва създаването на обект от тип **`IPAddress`**:

```
// IPv4 address - 212.30.23.111
// byte[] constructor does not work on .NET 1.1 for IPv4!!!
// Constructor by long
long addressNumber = (long) 111*256*256*256 + 23*256*256 +
  30*256 + 212;
IPAddress addr1 = new IPAddress(addressNumber);
// Parse method
IPAddress addr2 = IPAddress.Parse("212.30.23.111");
```

```
Console.WriteLine("addr1={0}, addr2={1}", addr1, addr2);

// IPv6 address - 20ac:103:de85:12:0:0:1:23f
// byte[] constructor
byte[] bytes2 = {0x20,0xac,0x1,0x3,0xde,0x85,0x0,0x12,0x0,
  0x0,0x0,0x0,0x0,0x1,0x2,0x3f};
IPAddress addr3 = new IPAddress(bytes2);
// Parse method
IPAddress addr4 = IPAddress.Parse("20ac:103:de85:12:0:0:1:23f");

Console.WriteLine("addr3={0}, addr4={1}", addr3, addr4);

// Output:
// addr1=212.30.23.111, addr2=212.30.23.111
// addr3=20ac:103:de85:12::1:23f, addr4=20ac:103:de85:12::1:23f
```

Забележете, че при IPv6 всеки байт в масива отговаря само на две шестнайсетични цифри, т.е. само на половината от всяка четирицифрена група. Другата особеност е, че в числото от тип **long**, в което се съхранява адресът, се редът на байтовете е обърнат, което би могло да доведе до объркване, ако не внимаваме.

Някои полезни методи и свойства

Класът **IPAddress** разполага с няколко удобни възможности, които бихме могли да използваме. Това са най-вече read-only полетата **Loopback**, **Any**, **None** и **Broadcast**, които ни предоставят няколко стандартни адреса във вида на инициализирани **IPAddress** обекти.

- **Loopback** е локалният адрес (127.0.0.1) на машината.
- **Any** (0.0.0.0) и **None** (255.255.255.255) са абстрактни адреси, които се използват при свързването на сокети, за да означат, че сървърът може да слуша за идваща връзка съответно от всеки един (**Any**) или нито един (**None**) адрес.
- Полето **Broadcast** (също 255.255.255.255) предоставя специален адрес в локалната мрежа. Ако изпратим IP пакет на този адрес, ще го получат всички, свързани в мрежата.
- Свойството **AddressFamily** показва дали адресът е от IPv4 или IPv6 тип.
- Методът **GetAddressBytes()** връща масив от байтовете на адреса. Резултатът от този метод може да се използва в конструктора **IPAddress(byte[])**, който вече споменахме.

Класът IPEndPoint

Класът **IPAddress** ни дава добра абстракция на един стандартен адрес в мрежата, но както вече обяснихме, за истинска връзка с обмен на данни се нуждаем и от даден порт. Пространството **System.Net** предлага класа **IPEndPoint** като абстракция на двойка (адрес, порт), която вече може да служи за създаването на връзката.

Обектите от класа **IPEndPoint** създаваме с един от двата конструктора **IPEndPoint(IPAddress address, int port)** или **IPEndPoint(long address, int port)**, като при втория подаваме като параметър числовата стойност на адреса (подобно на конструктора на **IPAddress**).

Свойствата **Address** и **Port** дават достъп съответно до адреса (във вид на обект от **IPAddress**) и порта на двойката. Свойството **AddressFamily** отново указва вида на адреса (IPv4 и IPv6).

Максималният и минималният допустим номер на порт можем да проверим с полетата **MaxPort** и **MinPort**. При все че **MinPort** обикновено има стойност 0, за нашите потребителски приложения е добре да резервираме портове с номера между 1024 и 65536. Както вече обяснихме, това е така, понеже останалите са резервирани за стандартни услуги.

Следният код демонстрира създаването на един **IPEndPoint** обект:

```
IPAddress address = IPAddress.Parse("212.30.23.111");
IPAddress addressV6 = IPAddress.Parse("20a:103:d5:12:0:0:1:2f");

IPEndPoint endpoint = new IPEndPoint(address,8080);
IPEndPoint endpointV6 = new IPEndPoint(addressV6,8081);

Console.WriteLine(
   "The endpoint ('{0}',{1}) has address type: {2}",
   endpoint.Address, endpoint.Port, endpoint.AddressFamily);
Console.WriteLine(
   "The endpoint ('{0}',{1}) has address type: {2}",
   endpointV6.Address, endpointV6.Port,
   endpointV6.AddressFamily);

// Output:
// The endpoint ('212.30.23.111',8080) has address type:
InterNetwork
// The endpoint ('20a:103:d5:12::1:2f',8081) has address type:
InterNetworkV6
```

Комуникация по TCP сокет с TcpClient

За създаването на една функционираща TCP връзка се нуждаем едновременно от сървърна и клиентска част. Сървърната част е необходима, за да приеме "повикването" от клиентската и да създаде връзка между двете

крайни точки. Обикновено сървърната програма поема и самата комуникация по връзката. При модела на TCP връзки, възприет в .NET, сървърният клас **TcpListener** служи само за установяване на връзката с клиентската част (чиято абстракция е класът **TcpClient**). След това се инициализира втора инстанция на клиентския клас, която да осъществи самата комуникация от страна на сървъра. Така на практика **TcpClient** върши основните задачи, свързани с преноса на данни по TCP връзката. Ще разгледаме първо неговите особености и начин на употреба, след което ще се спрем и на класа **TcpListener** и ще покажем как може да се създаде двуслойно мрежово приложение, едната част от което играе ролята на сървър, а другата – на клиент.

Създаване и свързване на TcpClient

За създаването на обект от класа **TcpClient** можем да подходим по два начина – да създадем несвързан клиент, който после да свържем чрез метода **Connect(…)**, или да създадем клиент, който още при инициализирането си да опита да се свърже с дадения сървър.

Несвързан клиент създаваме чрез конструктора **TcpClient()**. Той инициализира TCP сокет, който се обвързва с локалния мрежов интерфейс и със случайно избран от операционната система свободен порт. За да образуваме истинска TCP връзка, трябва да зададем и втория сокет, който може да бъде както на отдалечена машина, така и на локалната.

Задаването на другия край на връзката правим чрез метода **Connect(address, port)**. Той има три варианта, като в зависимост от данните, с които разполагаме, можем да зададем двойката **(address, port)** като:

- един аргумент от тип **IPEndPoint** – **Connect(IPEndPoint)**
- адрес като **IPAddress** обект и порт като число – **Connect(IPAddress, int)**
- адрес като низ и порт като число – **Connect (string, int)**

Последният вариант се използва най-често, тъй като не се налага да създаваме излишни обекти. Другите два са полезни, когато сме получили адреса или цялата двойка като резултат от друга операция.

Методът **Connect(…)** се опитва да осъществи връзка към указаната комбинация от адрес и порт. Ако това стане успешно, **TcpClient** обектът минава в свързано състояние и по него вече могат да се предават данни. В противен случай се хвърля **SocketException** с описание на грешката. Ето защо е добре да ограждаме извикването към метода **Connect(…)** в **try-catch-finally** блок, като във **finally** частта да затваряме сокета чрез метода **Close()**.

Методът **Close()** затваря създадения сокет и извършва действията по освобождаването на ресурсите, свързани с него. Както повечето подобни

методи в .NET, той се извиква автоматично при унищожаването на обекта, но винаги е добра идея да си подсигурим извикването му в случай на изключение, което възпрепятства автоматичното обръщение към метода.

Казаното дотук можем да илюстрираме със следния пример:

```
TcpClient client = new TcpClient();
try
{
   client.Connect("www.abv.bg", 80);
}
catch (SocketException se)
{
   Console.WriteLine("Could not connect: {0}", se.Message);
}
finally
{
   client.Close();
}
```

Променливата **client** декларираме извън **try** блока, за да можем да я използваме и след свързването.

Създаване на свързан TcpClient

Вторият вариант за свързване на клиентския обект е да използваме някой от конструкторите **TcpClient(IPEndPoint endpoint)** и **TcpClient(string address, int port)**. При тях отново се инициализира сокет свързан с локалния мрежов адрес и с произволен свободен порт на машината, но веднага след това новият **TcpClient** обект опитва да осъществи връзка с двойката адрес-порт, зададена от параметрите (съответно под формата на **IPEndPoint** обект или като низ и число). Както и при метода **Connect(…)**, ако връзка не може да се установи, се хвърля **SocketException**.

```
TcpClient client;
try
{
   client = new TcpClient("www.abv.bg",80);
}
catch (SocketException se)
{
   Console.WriteLine("Could not connect: {0}", se.Message);
}
finally
{
   if (client != null)
      client.Close();
}
```

При този код трябва да имаме предвид възможността конструкторът да предизвика **OutOfMemoryException** и кодът **client.Close()** във **finally** блока на свой ред да доведе до **NullPointerException**. Ето защо там проверяваме дали обектът от клиентския клас е бил създаден и едва тогава се обръщаме към метода **Close()**.

Създаване на прост TCP порт скенер – пример

Ще приложим наученото за създаването и свързването на сокетите от класа **TcpClient**, за да проверим състоянието на портовете на локалната машина. За целта последователно ще опитаме свързване с адрес 127.0.0.1 (локалния адрес на мрежовия интерфейс) на различни портове и ще проверяваме дали връзката е успешна. Ако попаднем в обработка на изключение тип **SocketException**, портът би трябвало да е затворен, в противен случай връзка може да се осъществи и портът е отворен. Разбира се това е прост пример и скенерът няма да има нито бързината, нито функционалността на истинските порт скенери, но все пак демонстрира използването на класа **TcpClient**.

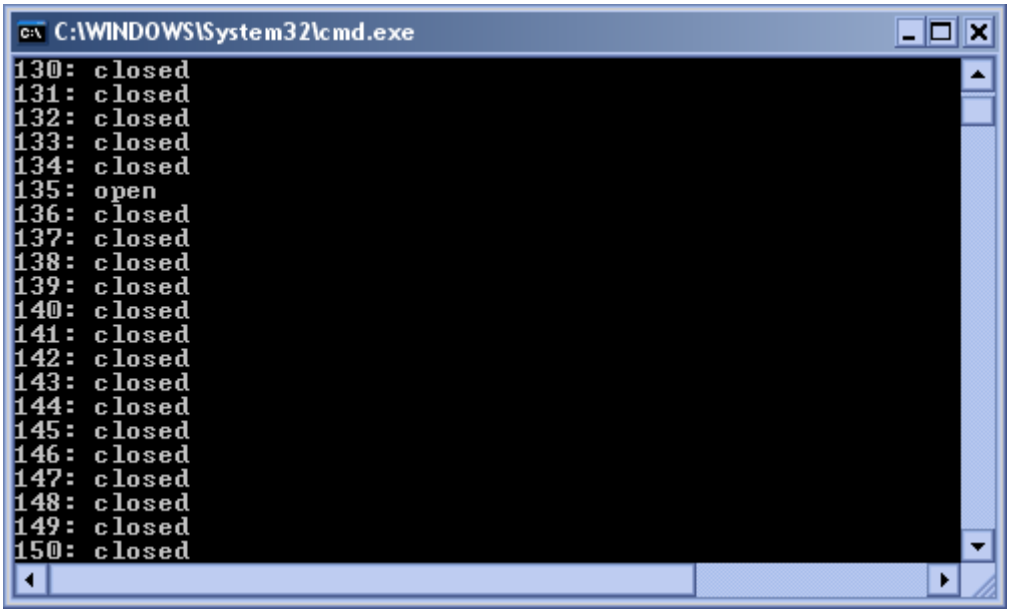
За изграждането на скенера изпълняваме следните стъпки:

1. Стартираме VS.NET и създаваме нов конзолен проект.
2. Въвеждаме кода на програмата, който е подобен на изложените по-горе примери. За локалния адрес използваме полето **IPAddress.Loopback**.

```
using System;
using System.Net;
using System.Net.Sockets;

class SimpleTcpScanner
{
  static void Main(string[] args)
  {
    // Scanning local TCP ports from 130 to 150
    for (int port = 130; port <= 150; port++)
    {
      TcpClient tcpClient = new TcpClient();
      try
      {
        // IPAddress.Loopback = "127.0.0.1"
        tcpClient.Connect(IPAddress.Loopback, port);
         Console.WriteLine("{0}: open", port);
      }
      catch (SocketException)
      {
        Console.WriteLine("{0}: closed", port);
      }
      finally
```

```
            {
                tcpClient.Close();
            }
        }
    }
}
```

3. Сега вече можем да стартираме програмата. Както се вижда от изхода, повечето портове, които сме опитали да отворим на тази машина, са затворени:



Предаване на данни по TCP сокет чрез TcpClient и NetworkStream

За да извършим комуникация и пренос на данни по създадената TCP връзка, трябва да извикаме метода **GetStream()** на класа **TcpClient**, след като сме свързали нашия сокет към някой сървър. Методът **GetStream()** връща обект от тип **NetworkStream**, чиито методи използваме за прехвърлянето на информация по мрежата. Ако **TcpClient** обектът не е свързан, методът **GetStream()** предизвиква изключение от тип **InvalidOperationException**. Следният код създава поток за писане в нашия TCP сокет:

```
TcpClient client = new TcpClient("www.abv.bg", 80);
NetworkStream stream = client.GetStream();
```

Класът **NetworkStream** наследява класа **Stream** и запазва свойствата и методите, характерни за всички потоци с някои особености. На практика

предаването и приемането на данни по TCP протокол се свежда до употребата на методите **Read(…)** и **Write(…)** – в .NET мрежовата връзка също се представя като абстракция чрез поток, в който може да се пише и да се чете.

Особености на NetworkStream

Мрежовите потоци се характеризират най-вече с това, че не позволяват произволен достъп до данните. Това е така, понеже данните идват на порции по TCP връзката и не е възможно да се знае точният им размер, нито да се избират данни на произволно място в потока. Ето защо четенето и писането в мрежовите потоци е само последователно. По тази причина свойството **CanSeek** винаги има стойност **false**, а използването на наследените свойства **Length** и **Position**, както и на методите **Seek** и **SetLength**, предизвиква **NotSupportedException**.

Друга особеност е, че затварянето на мрежовия поток с метода **Close()** обикновено не е необходимо – то се извършва при затварянето на прилежащия сокет (в случая **TcpClient**). Обратното обикновено не е вярно, но можем да използваме конструктора **NetworkStream(Socket socket, bool ownsSocket)**, за да създадем поток, обвързан със свързан сокет, така че затварянето на потока, да доведе и до затваряне на сокета. За целта на параметъра **ownsSocket** трябва да присвоим стойност **true**.

Конструкторите на **NetworkStream** използват класа **Socket**, за който още не сме говорили, но те ни дават възможност да извлечем поток от произволни свързани сокети с различни мрежови протоколи, стига те да имат стойност **SocketType.Stream** в свойството **SocketType**. За това ще стане дума отново малко по-късно. Полезно е да знаем, че при използването на тези конструктори можем да зададем и параметър от тип **FileAccess**, който да управлява и режима на достъп на потока (**Read**, **Write** или **ReadWrite**):

```
Socket socket = new Socket(AddressFamily.InterNetwork,
  SocketType.Stream, ProtocolType.Tcp);
socket.Connect(new IPEndPoint
  (IPAddress.Parse("127.0.0.1"), 135));
NetworkStream stream = new NetworkStream(socket,
  FileAccess.ReadWrite,true);
// FileAccess values require using the System.IO namespace
stream.Close(); // This also closes the socket
```

Ако сокетът не е свързан, или не е от поточен тип (**SocketType.Stream**), опитът за създаването на **NetworkStream** по този сокет води до **IOException**. Правата за четене и писане на потока могат да бъдат проверени чрез познатите от потоците свойства **CanWrite** и **CanRead**.

Както виждаме, класът **NetworkStream** предлага доста възможности и различни начини за действие и създаване. При все това, когато работим с

опростения клас **TcpClient**, най-удобно е да използваме именно метода **GetStream()** за получаване на поток, който е в състояние да изпълнява всички необходими за TCP връзката дейности.

Приемане на данни с NetworkStream

След като сме създали TCP сокет и сме получили достъп до поток, по който да извършваме трансфер на информация в този сокет, четенето (приемане) на данни можем да осъществим с метода **Read(byte[] buffer, int offset, int size)**. Параметърът **buffer** е масив, в който ще съхраняваме получените данни, а **size** указва колко байта да бъдат прочетени (най-много размера на масива). Параметърът **offset** указва от коя позиция нататък да започне записването на данни в масива. Това се използва, когато по някаква причина четем порции, по-малки от големината на буфера и следващата порция трябва да бъде записана от някаква следваща позиция нататък, за да не припокрие вече прочетените данни. Един пример за четене с метода **Read(…)** изглежда така (за яснота тук и в други примери спестяваме обработката на изключенията, каквато иначе не трябва да пропускаме):

```
TcpClient client = new TcpClient("www.abv.bg", 80);
NetworkStream stream = client.GetStream();
byte[] buffer = new byte[4096];
int bytesRead = stream.Read(buffer, 0, buffer.Length);
```

Методът **Read(…)** връща броя на прочетените байтове. Той не е задължително броят, указан от параметъра **size**, защото данните, изпратени от отсрещната страна, може да са по-малко. Ще обърнем внимание, че този метод блокира изпълнението на програмата – това означава, че последващите операции не се изпълняват, докато не бъдат прочетени някакви данни (както и при всички потоци в синхронно изпълнение). Ако например изпълним горния пример, ще се случи точно това – тъй като нашият клиентски сокет не е изпратил заявка към abv.bg по TCP връзката, а просто се е свързал, той няма да получи никакви данни оттам и изпълнението на програмата остава блокирано.

Това може да се избегне, като проверим стойността на свойството **DataAvailable** на класа **NetworkStream**, преди да изпълним **Read(…)**. Ако това свойство е **true**, то или в потока има данни за четене, при което методът **Read(…)** завършва, или сокетът е затворен от отсрещната страна, при което **Read(…)** отново завършва и връща стойност 0 (т.е. той не може повече нищо да прочете от затворения сокет):

```
TcpClient client = new TcpClient("www.abv.bg", 80);
NetworkStream stream = client.GetStream();
byte[] buffer = new byte[4096];
int bytesRead;
if (stream.DataAvailable)
```

```
bytesRead = stream.Read(buffer, 0, buffer.Length);
```

След тази преработка програмата завършва успешно при изпълнението на горния код.

Предаване на данни с NetworkStream

Изпращането на данни по връзката става с поточния метод **`Write(byte[] buffer, int offset, int size)`**. При него подаваме буфер в паметта, от който да се чете информацията, която трябва да бъде записана в потока. Параметърът **`offset`** показва от коя позиция нататък да се чете информация от буфера, а **`size`** – колко байта да бъдат записани от буфера в потока. Методът **`Write(…)`** не връща стойност – ако не съществува проблем на по-ниско ниво, например с мрежовата връзка, в потока винаги може да се запишат произволно количество данни, ето защо винаги се прави опит да се запишат всички указани от **`size`** байтове.

Особеност на мрежовото предаване на данни по TCP е, че самият протокол използва вътрешно буфериране при предаването на данни. Това означава, че ако запишем в потока информация, по-малка от размера на този вътрешен буфер, нямаме гаранция, че тя ще достигне до отсрещната страна. За тази цел винаги трябва да използваме метода **`Flush()`** след извикването на **`Write(…)`**. Това ни осигурява със сигурност изпращането на данните по връзката:

```
stream.Write(buffer, 0, buffer.Length);
stream.Flush();
```

Комуникация с TcpClient – пример

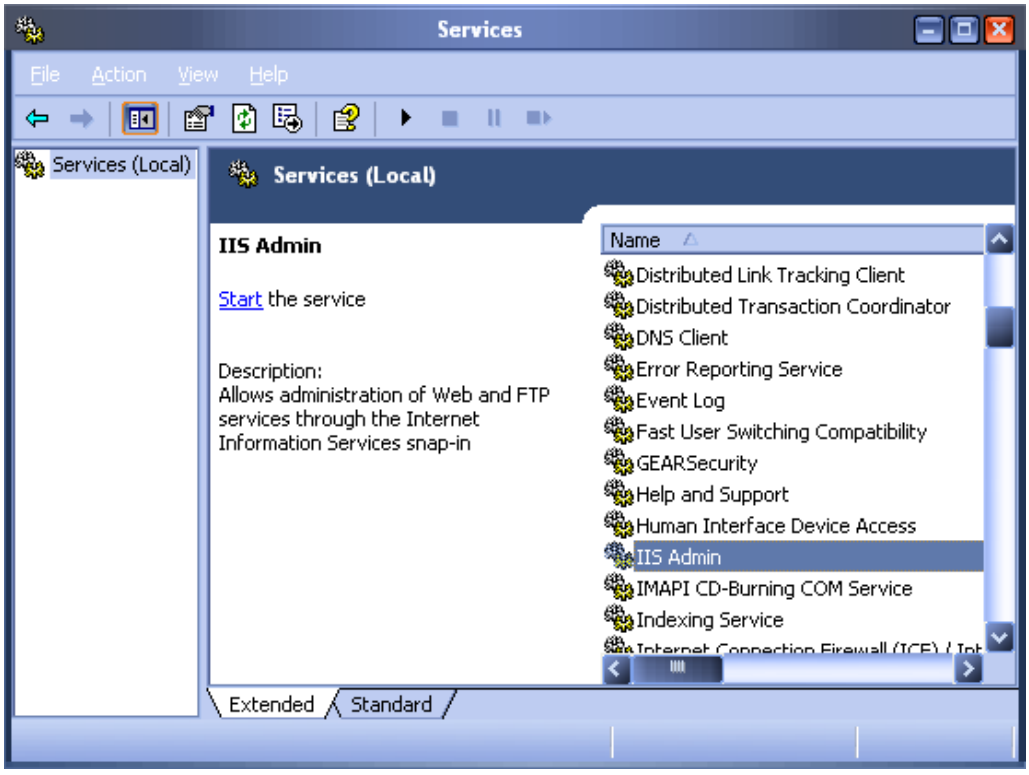
Ще демонстрираме едно напълно изградено просто клиентско приложение, реализирано с помощта на класа **`TcpClient`**. За да покажем комуникацията по TCP канала, имаме два варианта. Можем да използваме някой сървър в Интернет, а можем да използваме и локален сървър – сървъра **IIS (Internet Information Services)**, който се инсталира като част от Windows 2000, XP и 2003 Server.

Нашият пример ще работи по следния начин: първо създаваме инстанция на класа **`TcpClient`**, а после се опитваме да се свържем с избрания хост на порт 80, където обикновено слуша HTTP сървър. След това извличаме потока, в който ще предаваме данни в отворения сокет. Чрез метода **`Write(…)`** ще изпратим една заявка **HTTP GET**, с която да поискаме съдържанието на заглавната страница на домейна, с който се сме се свързали. После прочитаме отговора на сървъра като използваме метода **`Read(…)`** в цикъл докато резултатът от него стане 0 байта, което ще означава, че няма повече информация за четене. След това ще отпечатаме този отговор на екрана и ще завършим изпълнението на програмата.

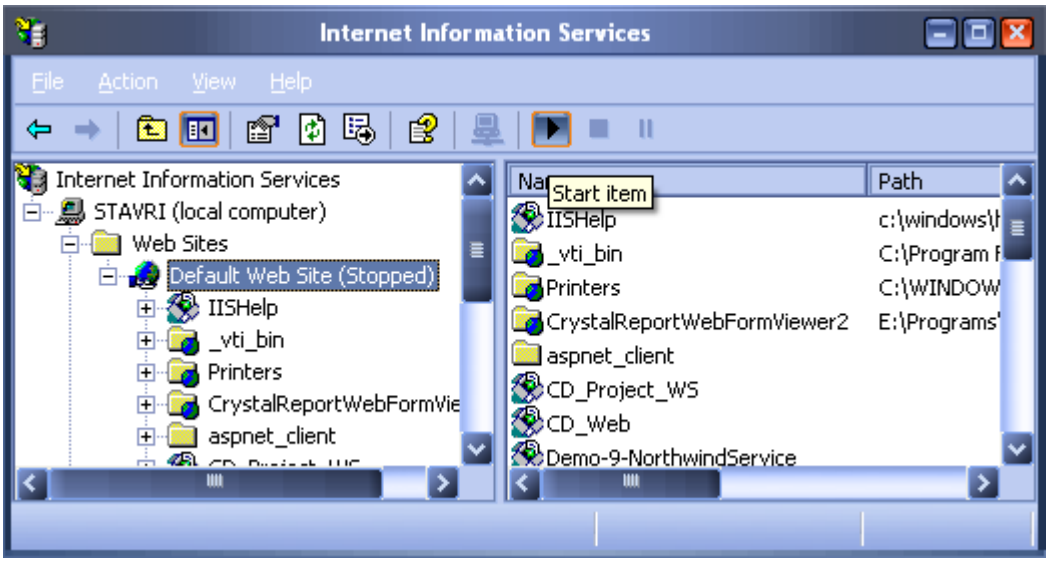
Необходимите стъпки за изграждането на примера са следните. Ако няма да използваме IIS, стъпките 1–3 може да се пропуснат.

1. Стартираме IIS, ако не е вече стартиран. Това може да се провери по следния начин – сървърът е стартиран, ако при поискването на адрес http://localhost/ в браузъра се зарежда страницата на IIS. Ако това е така, можем да минем на стъпка 4. В противен случай изпълняваме стъпки 2 и 3 за стартиране на сървъра.

2. В Windows XP можем да стартираме услугата, която отговаря за IIS, като отидем в **Control Panel | Administrative Tools | Services**. Избираме услугата **IIS Admin** от списъка, след което избираме етикета **Extended** под списъка на услугите и щракваме върху препратката **Start the service**:



3. Сега отваряме **Control Panel | Administrative Tools | Internet Information Services** и чрез навигация в дървото отляво последователно отваряме **Local Computer | Web Sites | Default Web Site**. След това трябва да щракнем върху бутона **Start Item** върху лентата с инструменти горе вдясно:



4. При стартиран IIS, вече можем да създадем нов конзолен проект във Visual Studio .NET.

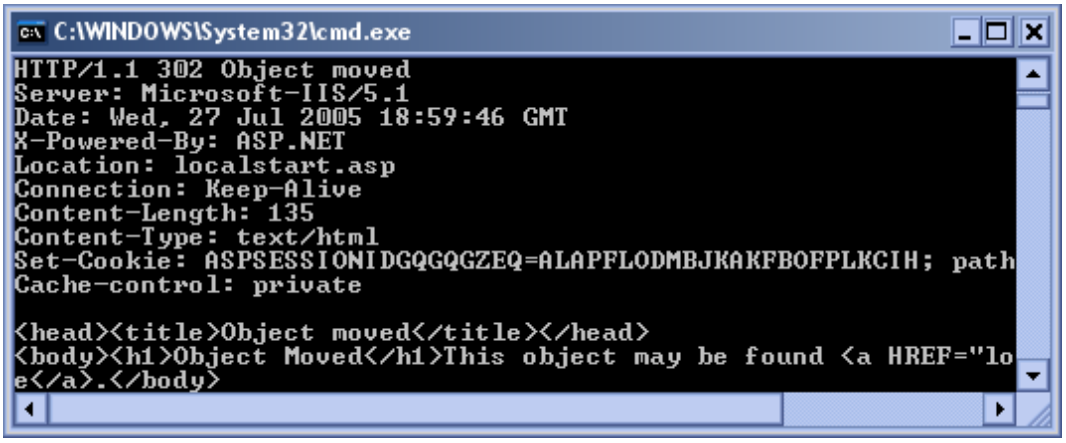
5. Въвеждаме кода на програмата. Обърнете внимание на употребата на класа **`StreamWriter`**, благодарение на който използваме метода **`WriteLine(…)`**:

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Text;

class TcpClientDemo
{
   const int RECEIVE_BUF_SIZE = 4096;

   static void Main()
   {
      // Connect to the server
      TcpClient tcpClient = new TcpClient();
      try
      {
         tcpClient.Connect("localhost", 80);
         // Possibly replace with "www.abv.bg"
      }
      catch (SocketException)
      {
         Console.WriteLine("Error: Unable to connect to the
         server.");
         Environment.Exit(-1);
      }
```

```
try
{
   NetworkStream ns = tcpClient.GetStream();
   using (ns)
   {
      // Send HTTP GET request to the Web server
      try
      {
         StreamWriter writer = new StreamWriter(ns);
         writer.WriteLine("GET http://localhost/ HTTP/1.0");
         // Possibly "GET http://www.abv.bg/ HTTP/1.0"
         writer.WriteLine();
         writer.Flush();
      }
      catch (IOException)
      {
         Console.WriteLine("Error: Cannot send request.");
         Environment.Exit(-1);
      }

      // Receive the HTTP answer from the server
      try
      {
         byte[] buf = new byte[RECEIVE_BUF_SIZE];
         while (true)
         {
            int bytesRead = ns.Read(buf, 0, buf.Length);
            if (bytesRead == 0)
            {
               // Server closed the connection
               break;
            }
            string data = Encoding.ASCII.GetString(buf, 0,
            bytesRead);
            Console.Write(data);
         }
      }
      catch (IOException)
      {
         Console.WriteLine("Error: Cannot read the server
         response.");
         Environment.Exit(-1);
      }
   }
}
finally
{
   // Close the connection to the server (if it is still
   open)
```

```
            tcpClient.Close();
        }
    }
}
```

6. Стартираме програмата. Очакваният изход е стандартният HTTP отговор, който трябва да съдържа тялото на поискания документ. Ако сме се свързали към IIS, отговорът ще ни казва, че обектът, който сме поискали, може да бъде намерен на адрес http://localhost/localstart.asp. Ако не нашето клиентско приложение, а някой истински браузър изпрати същата заявка, той ще бъде пренасочен именно към този адрес и ще изпрати автоматично нова заявка към него.



7. Ако сме използвали IIS, можем да видим какво става, когато няма връзка към посочения сървър. Нека спрем IIS подобно на описания в т. 2 и т. 3 начин, но със **Stop...** вместо със **Start...**, и отново стартираме програмата, ще получим също очакван резултат: "**`Error: Unable to connect to the server.`**", защото порт 80 на локалната машина е затворен и методът **`Connect(…)`** предизвиква изключение.

Настройки на TCP връзката чрез свойствата на TcpClient

Няколко свойства на класа **`TcpClient`** ни позволяват да контролираме различни параметри на комуникацията по TCP връзката. Внимателното им използване може да оптимизира трансфера на данни в някои по-особени случаи, например, когато използваме само малки пакети данни. Например чрез свойствата **`NoDelay`** и **`SendBufferSize`** може да се контролира кога точно изпратеното съобщение да напуска TCP буферите на операционната система. Повече информация за настройките на **`TcpClient`** може да се открие в MSDN документацията.

Изграждане на TCP сървър с TcpListener

Сега ще разгледаме класа **TcpListener**, с помощта на който можем да създадем сървърно приложение, с което да свържем вече показания TCP клиент. Ще напомним, че този клас служи единствено за приемане и инициализиране на връзка (connection) от страна на сървъра, а комуникацията по нея се осъществява чрез инстанции на класа **TcpClient** по начините, които вече разгледахме. Как точно се осъществява връзката между двата класа, ще разберем от следващите редове.

Създаване на TcpListener

Класът **TcpListener** има два конструктора, които са идентични – **TcpListener(IPEndPoint)** и **TcpListener(IPAddress, int)**. И двата конструктора задават адрес на локален мрежов интерфейс и на локален порт, на които сървърът да слуша за поискана TCP връзка. При първия конструктор двойката (адрес, порт) се задава чрез аргумент от тип **IPEndPoint**, а при втория се използва адрес под формата на **IPAddress** обект и цяло число за номер на порт. Възможно е да се използва и конструкторът **TcpListener(int)**, при който само задаваме порт, но се препоръчва вместо него да се използва някой от горните конструктори.

Ако не искаме да указваме конкретно на кой адрес ще "слуша" сървърът, можем да използваме конструкцията **IPAddress.Any** за задаване на адрес. Тогава при наличието на повече от един мрежов интерфейс на компютъра, сървърът ще слуша на всичките. Ако пък за номер на порт зададем числото 0, операционната система автоматично разпределя някой от свободните портове между 1024 и 65535.

Ако оставим порта да се избира автоматично, то стойността му можем да проверим след това чрез свойството **LocalEndpoint**. То връща стойност от абстрактния клас **EndPoint** и трябва да преобразуваме резултата по следния начин:

```
TcpListener listener = new TcpListener(IPAddress.Any,0);
listener.Start();
IPEndPoint endPoint = (IPEndPoint)listener.LocalEndpoint;
Console.WriteLine("Address: {0}; Port: {1}", endPoint.Address,
  endPoint.Port);
listener.Stop();
// Output:
// Address: 0.0.0.0; Port: 3108
```

Приемане на TCP връзки

В горния пример сме използвали метода **Start()**. Чрез този метод указваме на сървърния обект да започне да "слуша" за връзки на избрания (от нас или от операционната система) локален порт. Всяко клиентско прило-

жение, което се опита да се свърже с нашата машина на този порт, вече ще може да осъществи истинска TCP връзка.

Ако пристигнат повече заявки за връзки, те се нареждат в специална опашка и чакат да бъдат приети от сървъра. Приемането става чрез метода **AcceptTcpClient()**. Слушането за връзки продължава или докато бъде извикан методът **Stop()** (при което всички необработени заявки се губят), или докато опашката се запълни с максималното количество заявки, при което се предизвиква **SocketException**.

Методът **AcceptTcpClient()** приема първата чакаща заявка за връзка и връща като резултат инстанция на класа **TcpClient**. Чрез методите на този клас, които вече разгледахме, ние можем да управляваме връзката и да изпращаме и приемаме данни от страната на сървъра по същия начин, по който показахме при клиентското приложение. На практика двете страни извършват идентични действия при преноса на данни.

Описаните дотук действия можем да демонстрираме със следния кратък пример :

```
TcpListener listener = new
   TcpListener(IPAddress.Parse("127.0.0.1"), 2222);
listener.Start();
TcpClient client = listener.AcceptTcpClient();//blocks execution
listener.Stop();
NetworkStream stream = client.GetStream();
using (stream)
{
   StreamWriter writer = new StreamWriter(stream);
   writer.WriteLine("Hello!");
   writer.Flush();
}
client.Close();
```

Особености на метода AcceptTcpClient()

AcceptTcpClient() е блокираща операция. Ако в опашката няма чакаща заявка за връзка, изпълнението на програмата блокира до пристигането на такава заявка. За да избегнем този ефект, можем да проверяваме стойността, върната от метода **Pending()**. Ако тя е **true**, то има заявка за създаване на TCP връзка и можем да използваме **AcceptTcpClient()**, като сме сигурни, че той няма да блокира изпълнението.

Извикването на **AcceptTcpClient()** е възможно само за вече слушащ сървър (на който е бил извикан методът **Start()**), в противен случай се предизвиква изключение от тип **InvalidOperationException**). След завършването му най-горната заявка в опашката се премахва оттам и полученият обект от класа **TcpClient** поема изцяло комуникацията по тази връзка. Междувременно сървърът може да продължава да слуша на указания порт.

В частност горните думи означават също, че **TcpListener** обектът няма контрол върху сокетите, чрез които се реализират поетите при **AcceptTcpClient()** връзки и извикването на метода **Stop()** няма да ги затвори тези връзки. Програмистът сам трябва да се погрижи за затварянето на всяка отворена по този начин връзка.

Методът AcceptSocket()

Ще отбележим накратко, че освен **AcceptTcpClient()** съществува още един метод, който може да поеме заявка за TCP връзка. Това е методът **AcceptSocket()**, с който предаваме управлението на връзката на обект от тип **Socket**, а не на обект от тип **TcpClient**. Както ще видим по-късно, това ни дава повече гъвкавост и възможности.

Прост TCP сървър – пример

Ще разгледаме един по-цялостен пример, който реализира сравнително просто сървърно приложение чрез класовете **TcpListener**, **TcpClient** и **NetworkStream**. Нашият сървър ще слуша на порт 2222 на всички мрежови интерфейси и ще посреща всяка клиентска програма със заявка за TCP връзка. Сървърът се осведомява за името на клиента, изпращайки текстов ред по потока, създаден от връзката чрез класа **NetworkStream**, след което изпраща поздрав към него и приключва връзката. Приемането на клиентски връзки ще извършим в цикъл, така че нова връзка ще може да бъде осъществена едва когато е приключила текущата. Нашият сървър не може да обслужва повече клиенти едновременно и ще прави това последователно. Когато някой клиент въведе командата "**exit**" вместо име, цикълът ще приключи.

За да построим, стартираме и тестваме това приложение, трябва да извършим следните стъпки:

1. Стартираме VS. NET и създаваме нов конзолен проект.
2. Въвеждаме кода на програмата:

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Text;

class SimpleTcpServer
{
   const int LISTENING_PORT = 2222;
   const string EXIT_COMMAND = "exit";

   static void Main(string[] args)
   {
      IPEndPoint serverEndPoint =
```

```
        new IPEndPoint(IPAddress.Any, LISTENING_PORT);
      TcpListener server = new TcpListener(serverEndPoint);
      server.Start();
      Console.WriteLine("Simple TCP Server started listening on
        {0}...", server.LocalEndpoint);

      while (true)
      {
        TcpClient client = server.AcceptTcpClient();
        Console.WriteLine("Accepted client connection.");
        try
        {
          NetworkStream ns = client.GetStream();
          using (ns)
          {
            StreamWriter writer = new StreamWriter(ns);
            writer.Write("Enter your name: ");
            writer.Flush();
            StreamReader reader = new StreamReader(ns);
            string name = reader.ReadLine();
            if (name == EXIT_COMMAND)
            {
              break;
            }
            writer.WriteLine("Hello, {0}", name);
            writer.Flush();
          }
        }
        finally
        {
          client.Close();
          Console.WriteLine("Client connection closed.");
        }
      }

      server.Stop(); //This could also be in a try-finally block
      Console.WriteLine("Simple TCP Server stopped.");
    }
}
```

3. Стартираме програмата и я оставаме да работи в чакане на клиентски заявки за TCP връзка. Нашият сървър извежда съобщение "Simple TCP server started listening at 0.0.0.0:2222…".

4. За изпробването на проекта, ще трябва да използваме някакво клиентско приложение. Това може да е клиентът, който изградихме в предишния пример, но тук ще използваме вградения в Windows универсален клиент **telnet**. За целта отваряме **Command Prompt** от **Start менюто** в Windows или въвеждаме командата **cmd** в **Run**. На командния ред, при работещ още сървър, въвеждаме командата

`telnet localhost 2222`. Можем да видим в прозореца на нашето приложение, че е приета клиентската връзка и на екрана на telnet се показва искане за името на клиента. Въвеждаме някакво име, сървърът ни изпраща още един ред и връзката приключва:

5. Връзката на telnet е приключила, но нашият сървър продължава да слуша, докато въведем командата "exit" вместо име. Да опитаме да пуснем две клиентски сесии едновременно. Ще се убедим, че втората ще трябва да изчака завършването на първата и едва тогава получава съобщението "Enter your name:".
6. Сега можем да въведем и командата "exit" и да се убедим, че тя наистина приключва работата на сървъра.

Обслужване на много клиенти едновременно

Разбира се, сървър, който може да обслужва клиентите само последователно, не е особено полезен в практиката. Нуждаем се от механизъм, който да позволи на сървъра да се занимава независимо със създадените клиентски връзки. Този механизъм ни се осигурява от многонишковото програмиране (вж. темата за нишки и синхронизация).

Схемата за реализиране на едновременно обслужване на много клиенти е следната: използваме един клас, който служи за сървър и приема клиентските връзки по начина, описан в предишния пример. Обработката на връзката, която досега правихме в тялото на сървърния клас, сега изнасяме в отделен клас, обект от който се инициализира с получения при приемането на връзката **`TcpClient`**. Този клас осъществява комуникацията с клиента, след което затваря създадения TCP сокет.

В сървърния клас слушаме за заявки за TCP връзки и при получена такава, стартираме нова нишка, която да започне изпълнението си от главния метод на класа, който ще обработва клиентската връзка. По този начин обработката се осъществява паралелно и независимо от продължаващото в сървърния клас слушане за нови връзки, за които просто се създават нови нишки.

Главният метод в стартираната нишка има задачата да завърши комуникацията с клиента, да затвори сокета и накрая да прекрати собственото си действие, с което обработката на този клиент приключва.

Казаното дотук ще илюстрираме с едно вече завършено по-функционално приложение със сървърна и клиентска част.

Едновременно обслужване на клиенти с TcpListener – пример

С този пример ще покажем как се реализира схемата, която обяснихме по-горе. За сървърен клас ще ни служи класът **ThreadedTcpServer**, който е аналогичен на класа от предишния пример (отваря порт 2222 на локалната машина и слуша на него), но не обработва клиентската връзка. Тази логика ще изнесем в друг клас – **ClientThread**, в чийто конструктор ще подаваме като параметър обекта от класа **TcpClient**, който ще получим от **ThreadedTcpServer** при приемането на нова клиентска връзка.

Комуникацията с клиента осъществяваме в метода **ServeClient()** на класа **ClientThread**. До неговото изпълнение стигаме чрез създаване на нишка посредством класа **Thread**, след което в тялото му ще затворим отворените сокет и поток и ще приключим изпълнението на нишката. При възникване на проблем (изключение), програмата уведомява за грешка. Междувременно сървърът продължава да слуша за нови връзки (той няма да прекратява работата си при командата "exit", както беше при предишния пример).

Ще създадем и едно клиентско приложение в рамките на тази демонстрация, което да подобри нашето съществуващо приложение от първия пример. Тук ще използваме специален клас, който да осъществява само четенето и писането на данни, като двете дейности ще стартираме в отделни нишки. По този начин си осигуряваме безпроблемното едновременно извършване на двете дейности, които по принцип са блокиращи операции и могат да създадат проблем при невнимателно синхронизиране.

Създаване на многонишков TCP сървър – пример

Стъпките за реализация на сървърното приложение са следните:

1. Създаваме нов конзолен проект във VS .NET.
2. Въвеждаме кода на класа **ThreadedTcpServer**:

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Threading;

class ThreadedTcpServer
{
  const int LISTENING_PORT = 2222;
```

```
  public static void Main()
  {
    IPEndPoint serverEndPoint =
      new IPEndPoint(IPAddress.Any, LISTENING_PORT);
    TcpListener server = new TcpListener(serverEndPoint);
    server.Start();
    Console.WriteLine("Server started listening on {0}...",
      server.LocalEndpoint);

    while(true)
    {
      TcpClient client = server.AcceptTcpClient();
      ClientThread clientThread = new ClientThread(client);
      Thread thread = new Thread(
        new ThreadStart(clientThread.ServeClient));
      thread.Start();
    }
  }
}
```

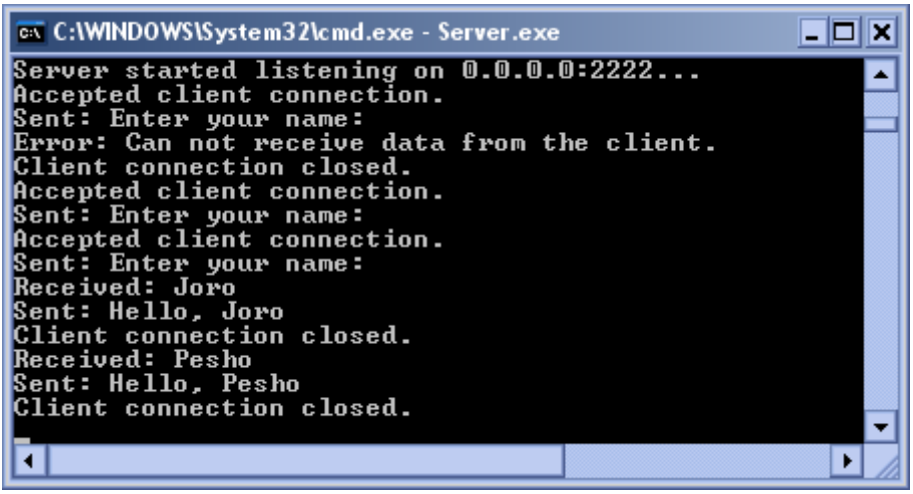
3. Добавяме и кода на класа **ClientThread**. Основният метод тук, както вече обяснихме, е **ServeClient()**, който извършва работата по комуникацията с клиента. Използваме помощните методи **SendText(string)** и **ReceiveLine()**, които съответно изпращат и приемат съобщенията до и от клиентското приложение. Те на свой ред използват класовете **StreamReader** и **StreamWriter** за по-удобни поточни операции. Отново обръщаме внимание на употребата на метода **Flush()**. Задали сме и стойност 10 секунди на свойството **ReceiveTimeout**. В случай на входно-изходни или мрежови грешки, ще предизвикаме едно потребителско изключение от клас **SendReceiveException**, който създаваме накрая:

```
class ClientThread
{
  const int SOCKET_TIMEOUT = 10 * 1000; // 10 seconds

  private TcpClient mTcpClient;
  private NetworkStream mNetworkStream;
  private StreamReader mReader;
  private StreamWriter mWriter;

  public ClientThread(TcpClient aTcpClient)
  {
    mTcpClient = aTcpClient;
    mTcpClient.ReceiveTimeout = SOCKET_TIMEOUT;
    mNetworkStream = mTcpClient.GetStream();
    mReader = new StreamReader(mNetworkStream);
    mWriter = new StreamWriter(mNetworkStream);
  }
```

```
public void ServeClient()
{
   Console.WriteLine("Accepted client connection.");
   try
   {
      SendText("Enter your name: ");
      string name = ReceiveLine();
      string response = "Hello, " + name + "\n";
      SendText(response);
   }
   catch (SendReceiveException sre)
   {
      Console.WriteLine(sre.Message);
   }
   finally
   {
      mNetworkStream.Close();
      mTcpClient.Close();
      Console.WriteLine("Client connection closed.");
   }
}

private void SendText(string aText)
{
   try
   {
      mWriter.Write(aText);
      mWriter.Flush();
      Console.WriteLine("Sent: {0}", aText.Trim('\n'));
   }
   catch (IOException ioex)
   {
      throw new SendReceiveException(
         "Error: Can not send data to the client.", ioex);
   }
}

private string ReceiveLine()
{
   try
   {
      string line = mReader.ReadLine();
      if (line == null)
      {
         throw new SendReceiveException(
            "Error: Connection closed by the client.", null);
      }
      Console.WriteLine("Received: {0}", line);
      return line;
```

```
      }
      catch (IOException ioex)
      {
         throw new SendReceiveException(
            "Error: Can not receive data from the client.", ioex);
      }
   }
}

class SendReceiveException : ApplicationException
{
   public SendReceiveException(String aMessage,
     Exception aCause) :   base(aMessage, aCause)
   {
   }
}
```

4. На този етап можем да стартираме приложението и отново да използваме telnet, за да се свържем с **`localhost`** на порт 2222. Както и в предишния пример, опитваме да осъществим едновременно няколко клиентски сесии. Този път това не е проблем за подобрения сървър. При тестването трябва да имаме предвид, че сме настроили сървъра да чака 10 секунди за въвеждане на вход от клиента, след което връзката се прекъсва (такъв случай се вижда на картинката). По желание можем да увеличим тази стойност.



Създаване на многонишков TCP клиент – пример

Нека сега създадем клиент за нашия сървър. Необходимо е клиентът да е способен едновременно да чете данни от сървъра и да чака вход от конзолата. При пристигане на данни от сървъра той ги отпечатва на конзолата, а прочетеното на конзолата изпраща към сървъра. По този начин симулира поведението на стандартния **`telnet`** клиент в Windows.

Ето стъпките за създаването на клиентското приложение:

1. Създаваме нов конзолен проект във VS .NET, където ще реализираме клиентско приложение.
2. Въвеждаме кода на класа **Client**. При реализацията му, както вече споменахме, ще създадем две нишки, едната от които ще чете, а другата ще пише. И двете започват в главния метод на класа **TextTransmitter**, който ще създадем след малко, като инициализират негови инстанции с различни потоци. Нишката, която ще чете от потребителя и ще праща към сървъра използва за четене потока **Console.In**, а за писане – създадения от мрежовия поток **StreamWriter**. Обратно, нишката, която чете от сървъра и извежда на екрана, ще използва за четене извлечения мрежов поток, обвит с инстанция на **StreamReader**, а ще пише в конзолния поток **Console.Out**. Върху нишките се изпълнява методът **Join()**, който, както знаем от главата за многонишково програмиране, кара текущата нишка да изчака приключването на тези две нишки, преди да може да завърши:

```
using System;
using System.Net;
using System.Net.Sockets;
using System.IO;
using System.Threading;

class Client
{
   const string SERVER_HOST = "127.0.0.1";
   const int SERVER_PORT = 2222;

   static void Main(string[] args)
   {
      TcpClient tcpClient = new TcpClient();
      try
      {
         tcpClient.Connect(SERVER_HOST, SERVER_PORT);
      }
      catch (SocketException)
      {
         Console.WriteLine("Can not connect to the server.");
         Environment.Exit(-1);
      }

      NetworkStream ns = tcpClient.GetStream();
      using (ns)
      {
         StreamReader reader = new StreamReader(ns);
         TextTransmitter serverToConsole =
            new TextTransmitter(reader, Console.Out);
```

```
        Thread serverToConsoleThread = new Thread(
          new ThreadStart(serverToConsole.Transmit));
        serverToConsoleThread.Start();

        StreamWriter writer = new StreamWriter(ns);
        TextTransmitter consoleToServer =
          new TextTransmitter(Console.In, writer);
        Thread consoleToServerThread = new Thread(
          new ThreadStart(consoleToServer.Transmit));
        consoleToServerThread.Start();

        serverToConsoleThread.Join();
        consoleToServerThread.Join();
      }
    }
}
```

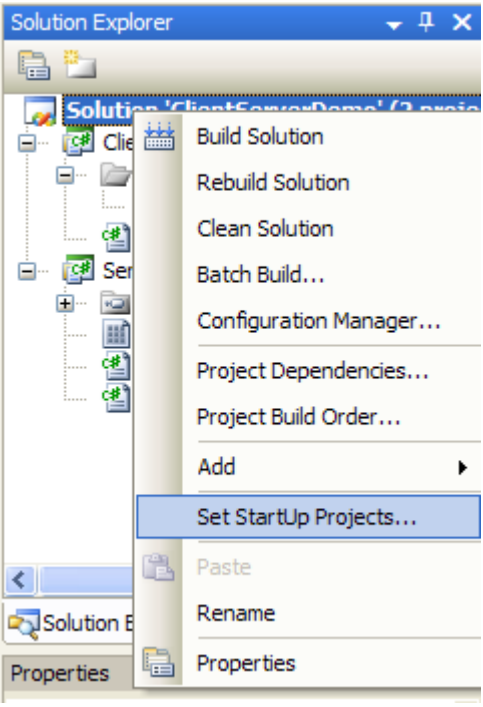
3. Накрая създаваме и класа **`TextTransmitter`**. Той е общ клас, който приема в конструктора си един поток за писане и един за четене, след което в основния си метод **`Transmit()`** използва безкраен цикъл, за да чете от първия поток и да записва прочетеното във втория. По този начин нишките, които използват конзолата за различни цели, не си пречат и комуникацията се осъществява гладко:

```
class TextTransmitter
{
  const int BUFFER_SIZE = 1024;

  private TextReader mReader;
  private TextWriter mWriter;

  public TextTransmitter(TextReader aReader, TextWriter aWriter)
  {
    mReader = aReader;
    mWriter = aWriter;
  }

  public void Transmit()
  {
    char[] buf = new char[BUFFER_SIZE];
    while (true)
    {
      int charsRead = mReader.Read(buf, 0, buf.Length);
      if (charsRead == 0)
      {
        // End of the stream reached --> the socket is closed
        Environment.Exit(-1);
      }
      mWriter.Write(buf, 0, charsRead);
      mWriter.Flush();
```

```
        }
    }
}
```

4. Сега вече сме готови да стартираме нашия клиент. За целта стартираме и сървъра (ако сме го изгасили) и проследяваме как можем едновременно с няколко клиента да получаваме информация. След познатото вече въвеждане на името и върнатия поздрав, връзката се прекратява от страна на сървъра и клиентската програма приключва изпълнението си.

Двата проекта можем да стартираме в отделни инстанции на Visual Studio .NET, а можем и да стартираме направо компилираните им изпълними файлове. Друг вариант е да обединим двата проекта в един solution на средата и да ги стартираме заедно. Това можем да направим като в отворен solution добавим последователно двата проекта чрез командата **File | Open | Project/Solution...** След това в **Solution Explorer** активираме командата **Set StartUp Projects...** и в диалоговия прозорец избираме опцията **Mulltiple startup projects**.



Комуникация по UDP с UdpClient

Както накратко обяснихме в началните бележки, комуникацията по UDP протокола се различава значително от тази по TCP. Втората (която досега разглеждахме) разчита на абстрактни "комуникационни канали", по които поточно да се предават данните. Там се извършва контрол на получената

информация, така че да няма опасност от загуба на данни. Това е по-сигурно, но се осъществява по-сложно и по-бавно. Ето защо в някои случаи е по-удобно да се използва друг тип комуникация.

При UDP обменът на данни е на "пакетен" принцип. Там няма комуникационен канал и ясно определена връзка (socket connection). Всяка от страните изпраща пакети от данни (**datagram packets**) на адреса на другата, без да се интересува дали те са стигнали, или не. Както вече посочихме, въпреки ненадеждността си, този тип обмен на данни е по-бърз от TCP, защото няма нужда да се поддържа комуникационен канал. UDP се предпочита, когато скоростта има по-голямо значение за приложението от сигурността и реда на пристигане на данните.

Понеже с UDP не се налага създаване и поддръжка на някаква връзка, няма и нужда от съответния сървърен клас, както при TCP комуникацията. Всички действия по обмена на данни чрез UDP се извършват в .NET от класа **`UdpClient`**, който ще разгледаме на следващите редове.

Конструктори на UdpClient

Класът има няколко конструктора, които отговарят на съответните конструктори на **`TcpClient`**. **`UdpClient(IPEndPoint LocalEP)`** ни позволява да създадем инстанция на **`UdpClient`**, свързана с даден локален UDP сокет (локален адрес и порт). При **`UdpClient(int port)`** указваме само порта, а при **`UdpClient()`** – оставяме операционната система да избере и него. Платформата .NET предлага и варианти на тези конструктори, при които се указва типът на стоящия отдолу протокол (IPv4 или IPv6), но това не е задължително. Конструкторът **`UdpClient(string host, int port)`** ще разгледаме след малко.

Задаване на отдалечен сървър по подразбиране

След като сме създали инстанция на **`UdpClient`** и сме я свързали с някакъв локален адрес, от който ще изпращаме datagram пакетите, вече сме готови да започнем обмена на данни. Първо обаче можем да зададем отдалечен сървър по подразбиране. Това не е задължително, но ни предлага удобството да не указваме сървъра при всяко изпращане на данни.

Задаването на отдалечен сървър по подразбиране може да стане по два начина. Единият е да използваме направо гореспоменатия конструктор **`UdpClient(string host, int port)`**. За разлика от останалите разгледани конструктори, той не задава локален адрес и порт (те се избират от операционната система), а задава именно отдалечен сървър.

Другият начин е чрез метода **`Connect(…)`** в някой от вариантите му – **`Connect(IPEndPoint)`**, **`Connect(IPAddress, int)`** и **`Connect(string, int)`**. При всичките задаваме отново адрес и порт на отдалечения сървър, на който ще пращаме пакети.

Независимо по кой начин сме задали сървъра, оттук нататък можем да пращаме и да получаваме данни само към и от него. Ако опитаме да укажем друг сървър на метода **Send(…)**, ще получим изключение. Сървърът по подразбиране може да се смени отново с метода **Connect(…)**.

Изпращане на UDP пакети – метод Send(...)

Метода **Send(…)** използваме, за да изпращаме пакети (datagrams) по UDP протокола. Ако сме задали отдалечен сървър по подразбиране, то чрез следния код можем да изпратим към този сървър низа "Hello, world":

```
UdpClient client = new UdpClient("127.0.0.1", 2222);
string hello = "Hello, world";
byte[] data = Encoding.ASCII.GetBytes(hello);
client.Send(data, data.Length);
client.Close();
```

Тук използваме варианта **Send(byte[], int)**. При него подаваме масив от байтове, които трябва да се изпратят с UDP пакета, както и втори параметър, указващ колко байта от масива да се изпратят. Обърнете внимание на начина, по който преобразуваме нашия низ в масив от байтове чрез методите на класа **Encoding**. Това е стандартната практика при изпращане на текстови данни.

Метода **Send(…)** можем да използваме и като зададем адрес (и порт), на който да се пращат данните. Както вече казахме, това може да стане само ако не сме задали вече такъв по подразбиране. Адресът подаваме като допълнителен параметър под формата на **IPEndPoint** или като низ и число за порт – **Send(byte[], int, string, int)**. В примера можем да видим първия вариант с **IPEndPoint**:

```
UdpClient client = new UdpClient();
string hello = "Hello, world";
byte[] data = Encoding.ASCII.GetBytes(hello);
IPEndPoint ep = new IPEndPoint("127.0.0.1", 2222);
client.Send(data,data.Length, ep);
client.Close();
```

Основното, което трябва да запомним за метода **Send(…)**, е че той не ни гарантира, че нашите данни са получени. Ако такова потвърждение е важно за нашата програма, трябва или да използваме TCP комуникация, или да разработим някакъв начин за нотификация за получени пакети от отсрещната страна.

Не бива да забравяме след приключване на обмена на данни да извикаме и метода **Close()**, за да затворим сокета.

Получаване на UDP пакети – метод Receive(...)

За разлика от **Send(…)**, този метод има само един вариант – **Receive(ref IPEndPoint sender)**. С него слушаме на локален адрес и порт за получени UDP пакети. Тази операция е блокираща и изпълнението на програмата спира, докато не получим пакет на отворения UDP порт, затова трябва да внимаваме с нея. Локалният адрес се задава при конструкторите, както видяхме, или пък се избира автоматично от операционната система, в случай че не е зададен.

Параметърът **sender** се предава по референция и в него се запазва информация за адреса и порта на сървъра, от който сме получили UDP пакета.

Резултатът от метода **Receive(…)** е масив от данни тип **byte**. Обикновено се налага да го преобразуваме до друг клас, чрез който да можем да работим с получените данни. Например низ можем да получим (обратно на горния пример) чрез метода **Encoding.GetString(byte[])**.

Следните редове код демонстрират използването на метода **Receive(…)**:

```
IPEndPoint listener = new IPEndPoint(IPAddress.Any, 1111);
UdpClient udpListener = new UdpClient(listener);
IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);
byte[] packet = udpListener.Receive(ref sender);
Console.WriteLine("{0} bytes received from {1}",
  packet.Length,sender);
Console.WriteLine("String rappresentation: {0}",
  Encoding.ASCII.GetString(packet));
```

Ако се опитаме да изпробваме този пример, ще се сблъскаме с проблема, за който вече споменахме. Изпълнението на програмата спира при метода **Receive(…)**, защото чака да получи някакъв UDP пакет. За да изпробваме приложението, ще ни трябва и друга програма, която да изпраща такива пакети. Това ще покажем в следващите редове.

Комуникация с UdpClient – пример

Ще разгледаме две приложения, използващи **UdpClient** класа, едното от които ще изпълнява ролята на приемник на информацията, а другото ще изпраща пакети. За целта трябва да подсигурим, че двете инстанции на **UdpClient** ще използват едни и същи порт и интерфейс, като за едната това ще бъде локален адрес (на който да слуша), а за другата – отдалечен (на който да праща). Ето как правим това.

1. Стартираме VS .NET студио и създаваме нов конзолен проект.
2. Въвеждаме кода на сървърния клас (който слуша за UDP пакетите). Обвързваме го с локалния адрес и порт 1111 чрез конструктора на **UdpClient**, след което изпълняваме метода **Receive(…)** в цикъл:

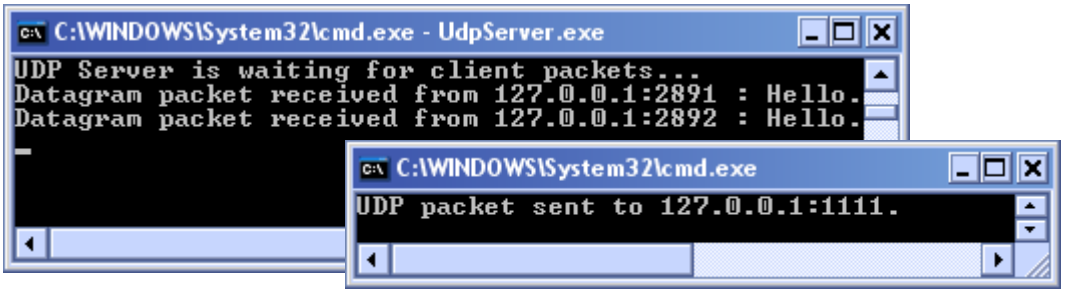
```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class UdpServerDemo
{
   const int LISTENING_PORT = 1111;

   public static void Main()
   {
      IPEndPoint serverEndPoint =
        new IPEndPoint(IPAddress.Any, LISTENING_PORT);
      UdpClient udpServer = new UdpClient(serverEndPoint);
      Console.WriteLine(
        "UDP Server is waiting for client packets...");

      while(true)
      {
         IPEndPoint senderEP = new IPEndPoint(IPAddress.Any, 0);
         byte[] packet = udpServer.Receive(ref senderEP);
         string message = Encoding.ASCII.GetString(packet);
         Console.WriteLine(
           "Datagram packet received from {0}:{1}.",
            senderEP, message);
      }
   }
}
```

3. Стартираме приложението и го оставяме да работи. Отваряме нова инстанция на Visual Studio .NET и създаваме нов проект, в който въвеждаме кода на клиентския клас. Клиентската програма трябва да използва същия порт за отдалечен сървър. При сървърната използвахме за адрес **`IPAddress.Any`**, но тук трябва да укажем точен адрес, на който да пращаме. Понеже сървърът върви локално, указваме локалния loopback адрес "127.0.0.1":

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class UdpClientDemo
{
   const string SERVER_HOST = "127.0.0.1";
   const int SERVER_PORT = 1111;

   public static void Main()
   {
      string welcomeMsg = "Hello";
```

```
        byte[] data = Encoding.ASCII.GetBytes(welcomeMsg);
        UdpClient udpClient = new UdpClient(SERVER_HOST,
          SERVER_PORT);
        udpClient.Send(data, data.Length);
        Console.WriteLine("UDP packet sent to {0}:{1}.",
          SERVER_HOST, SERVER_PORT);
    }
}
```

4. Сега стартираме и клиента и наблюдаваме ставащото в прозореца на сървъра. Там се извежда съобщение, че е получен UDP пакет, както и неговото съдържание във вид на текстов низ. Клиентът приключва действието си, а сървърът продължава да слуша. Можем да стартираме още веднъж клиентското приложение и ще наблюдаваме същия резултат. Сървърът ще е активен, докато прекъснем ръчно изпълнението му.



Сокети на по-ниско ниво – класът Socket

Класовете **`UdpClient`**, **`TcpClient`** и **`TcpListener`** обикновено са достатъчни за целите на програмирането с тези два протокола. За по-гъвкаво използване на сокетите можем обаче да използваме вместо тях класа **`Socket`**. Той предлага много повече възможности и всъщност разгледаните вече класове са негови наследници, специализирани за работа с даден протокол.

Класът **`Socket`** реализира абстракция на сокет в най-общ смисъл, съобразно функционалността, описана в интерфейса "Berkeley sockets". Това е набор от няколко дефинирани общи операции, които позволяват мрежовата комуникация по произволен мрежов протокол. На следващите редове ще разгледаме накратко как можем да използваме класа **`Socket`** за програмиране с произволен тип сокети на по-ниско ниво, което разширява гъвкавостта и възможностите на нашето приложение; както и проблемите, които може да срещнем, когато се лишим от тясната специализация на **`TcpClient`**, **`TcpListener`** и **`UdpClient`**.

Създаване на Socket обекти и тип на сокета

Конструкторът на класа изглежда ето така – **`Socket(AddressFamily, SocketType, ProtocolType)`**. Тук се подават три параметъра, всеки от ко-

ито е от специален изброен тип. Заедно трите определят вида на сокета. Класът **Socket** поддържа комуникация чрез голямо количество протоколи (TCP, UDP, IP, IPv6, ICMP, IGMP, IPX и др.) и именно параметрите, зададени при инициализацията, указват кой от тях ще се използва.

AddressFamily е изброен тип, който указва начина на представяне на адресите в комуникацията. За нашите цели ще използваме стойността **AddressFamily.InterNetwork**. Тя ни позволява да използваме IP адреси (да си припомним, че свойството **AddressFamily** на обектите от класа **IPEndPoint** връща точно тази стойност – всъщност по-общият клас **EndPoint** съществува именно защото с класа **Socket** можем да използваме и други типове адресиране).

SocketType е друг изброен тип, чиито стойности определят вида на сокета. Различните видове сокети имат различни характеристики. Някои поддържат връзка между страните, други – не. Отделно някои предават данните в пакети с точно определени граници, докато други използват поточна комуникация, в която границите между пакетите се губят. За различните протоколи, които искаме да използваме, трябва да изберем подходящия вид сокет. Ако не направим това коректно, ще получим **SocketException** при извикването на конструктора. Ние ще използваме стойностите **SocketType.Stream** за TCP (поточна комуникация с връзка) и **SocketType.Dgram** за UDP (пакетна комуникация без връзка). Други възможни типове са **SocketType.Raw** (директно предава IP пакети без допълнителна обработка, например за протоколи ICMP или IGMP), **SocketType.Seqpacket** (пакетна комуникация с връзка) и др.

Накрая, параметърът от изброения тип **ProtocolType** определя самия протокол, който .NET ще използва, за да предава съобщенията. Както вече казахме, комбинацията от **ProtocolType** и **SocketType** трябва да е коректна. За **ProtocolType.Tcp** трябва да сме задали **SocketType.Stream**, а за **ProtoclType.Udp** – **SocketType.Dgram**.

Следният код създава един поточен сокет с връзка:

```
Socket socket = new Socket(AddressFamily.InterNetwork,
   SocketType.Stream, ProtocolType.Tcp);
```

Основни операции с класа Socket

Нека сега разгледаме кои са операциите, дефинирани от Berkeley Sockets Interface, и чрез кои методи те се реализират в .NET Framework.

Както знаем, един сокет може да изпълнява ролята на сървър или на клиент. От страната на сървъра, всеки сокет трябва да изпълни няколко действия. Ако сокетът използва връзки, той трябва да се обвърже с някой порт на локалната система, да го отвори и да започне да слуша за идващи заявки за връзка, след което да приема връзката и да изпраща и приема

данни от клиентския сокет. Ако използваният протокол не изисква връзки, се иска само сървърът да се обвърже с някой порт.

Сокети с връзка по TCP

Ще разгледаме първо операциите, които трябва да се извършат от сървъра, а после и тези от клиента.

Класът Socket като сървър

Свързването с локален порт става чрез метода **`Bind(IPEndPoint localEP)`**. Подава му се параметър, който съдържа локален мрежов адрес и номер на порт. Както знаем, можем да използваме **`IPAddress.Any`** за адрес, както и 0 за номер на порт, ако искаме той да се избере от операционната система.

След като сме извикали **`Bind(…)`**, започваме да слушаме за връзки чрез метода **`Listen(int backlog)`**. Параметърът **`backlog`** е число, което указва колко заявки за връзки могат да бъдат задържани от операционната система в опашка, докато приключи първата. При избирането на тази стойност трябва да помним, че голяма опашка може да забави действието на програмата.

Приемането на клиентски връзки става чрез метода **`Accept()`**. Той ни връща обект от тип **`Socket`**, който описва новата връзка и чрез който можем да комуникираме с клиентската страна. Върху този сокет не можем отново да прилагаме **`Bind(…)`** и **`Listen()`**. Можем обаче да получим информация за адреса, от който идва връзката, чрез свойството **`RemoteEndPoint`**. Трябва да имаме само предвид, че това свойство връща резултат от общия клас **`EndPoint`** и трябва да го преобразуваме до **`IPEndPoint`**.

Както се вижда, действията, които извършвахме с **`TcpListener`**, силно напомнят тези с класа **`Socket`**. Там методите **`Bind(…)`** и **`Listen()`** бяха обединени в **`Start()`** и конструктора, а методът **`AcceptTcpClient()`** позволяваше да инициализираме комуникационния сокет направо със специализирания клас **`TcpClient`**.

Следният пример демонстрира описаните операции:

```
Socket socket = new Socket(AddressFamily.InterNetwork,
  SocketType.Stream, ProtocolType.Tcp);
socket.Bind(new IPEndPoint(IPAddress.Any, 2222));
socket.Listen(10);
Console.WriteLine("Waiting for a client...");
Socket client = socket.Accept();
IPEndPoint clientEP =
  (IPEndPoint)client.RemoteEndPoint;
Console.WriteLine("Connected with {0}:{1}",
  clientEP.Address, clientEP.Port);
```

```
// Communication with the client goes here

client.Close();
socket.Close();
```

Класът Socket като клиент

Работата със **Socket** от страната на клиента е още по-лесна. Отново трябва да създадем нужния ни сокет със **SocketType.Stream** и **ProtocolType.Tcp**, след което, вместо да прилагаме методите **Bind(…)** и **Listen(…)**, трябва да се свържем със сървъра чрез метода **Connect(remoteEP)**. Като аргумент подаваме обект от типа **IPEndPoint**, за да опишем адреса и порта, с които искаме да направим връзка. След завършването на метода **Connect(…)** вече сме създали свързан сокет и можем да го използваме за предаване на данни между клиента и сървъра.

Понеже връзка по мрежата не винаги може да се осъществи, при създаването на клиентската програма е хубаво да изпълняваме **Connect(…)** в **try-catch** блок. Така, ако възникне проблем при връзката, е достатъчно да прихванем предизвикания **SocketException** и да уведомим потребителя, че програмата не може да се свърже със сървъра.

След приключването на комуникацията трябва да извикаме последователно методите **Shutdown(SocketShutdown)** и **Close()**. Първият подсигурява, че всички данни, чакащи по връзката, са предадени и приети успешно, преди тя да бъде прекратена. Аргументът му е от изброения тип **SocketShutdown**, като освен ако не искаме да имплементираме някакво специално поведение, трябва да подадем **SocketShutdown.Both**. Методът **Close()** освобождава ресурсите, свързани със сокета, и го затваря. Той трябва да се извика и от сървърната страна. В примерите ще покажем как точно става това.

Как да предаваме данни между клиента и сървъра?

Данни по TCP връзката можем да предаваме чрез методите **Send(…)** и **Receive(…)**. Те имат по няколко форми, като в най-простия им вид просто трябва да подадем един масив от тип **byte[]**, който да служи за източник или приемник на предаваните данни. По желание можем да указваме допълнително колко точно байта да се изпратят или приемат по връзката, а чрез вариантите **Send(byte[], int, SocketFlags)** и **Receive(byte[], int, SocketFlags)** можем да указваме и различни опции за комуникацията, които са стойности на изброения тип **SocketFlags**. Методът **Receive(…)** връща като резултат броя получени по връзката байтове. Когато този резултат е 0, връзката е била прекратена от отсрещната страна. Можем да използваме този факт като условие за прекратяване на цикъла, в който обикновено ще предаваме и приемаме данните.

Обърнете внимание, че методите **Send(…)** и **Receive(…)** не приемат аргументи, указващи с кого се провежда комуникация. В момента на извикването им сокетът вече трябва да е свързан – или да е бил извикан методът **Connect(…)** от клиентската страна, или да е получен като резултат от метода **Accept()** за сървърната страна. Следните примери за двата вида сокети демонстрират начина на употреба на **Send(…)** и **Receive(…)** след като вече са изпълнени по свързване от горните примери:

```
// Server code
string welcome = "Welcome to my test server";
data = Encoding.ASCII.GetBytes(welcome);
client.Send(data, data.Length, SocketFlags.None);

// Client code
Socket socket = new Socket(AddressFamily.InterNetwork,
   SocketType.Stream, ProtocolType.Tcp);
try
{
   socket.Connect(new IPEndPoint(
      IPAddress.Parse("127.0.0.1"), 2222));
}
catch (SocketException e)
{
   Console.WriteLine("Unable to connect to server.");
   return;
}

int recv = socket.Receive(data);
stringData = Encoding.ASCII.GetString(data, 0, recv);
Console.WriteLine(stringData);
```

Ако и двете страни се опитат по едно и също време да изпращат или да чакат данни, ще се стигне до ситуация, в която и клиентът, и сървърът ще останат завинаги блокирани, понеже тези операции са блокиращи. Ние сами трябва да подсигурим, че при изпращането и приемането на данни страните се редуват.

Поточна комуникация

Данни с класа **Socket** можем да предаваме и поточно. За целта използваме класа **NetworkStream**, който вече разгледахме. Той има конструктор, който приема като параметър обект от тип **Socket** и лесно можем да го създадем от нашия сокет. Поточното предаване на данни има някои предимства и по желание можем да използваме него и методите му **Read(…)** и **Write(…)**, които по-горе обяснихме подробно. Свойството **DataAvailable** ни позволява във всеки един момент да проверим дали има данни за четене от потока.

Още повече функционалност можем да придобием, ако създадем обекти на класовете **StreamWriter** и **StreamReader**. Те се инициализират чрез

NetworkStream обекта и ни дават възможност да изпращаме и четем текстови данни от потока. Методите **ReadLine()** и **ReadToEnd()** ни позволяват да четем данни до края на един текстов ред или до края на целия поток. Обърнете внимание, че те ще върнат **null**, ако няма данни в потока, а методът **Read(…)** ще върне 0 като брой прочетени байтове, но за разлика от метода **Receive(…)** на класа **Socket**, това не означава, че връзката е затворена. За да проверим дали това е така, трябва да ограждаме извикването на метода в **try-catch** блок и да прихващаме **IOException**, предизвикването на който най-вероятно е било свързано с преустановяване на връзката.

Класът **StreamWriter** има съответния метод **WriteLine(…)**, който директно изпраща текстов ред по сокета. Нещо, което трябва да запомним при използването на този метод, както и на всички поточни методи за писане, е винаги да извикваме метода **Flush()** след тях.

Винаги извиквайте метода `Flush()` след като приключите с писането в мрежов поток! По този начин осигурявате, че данните със сигурност са изпратени по връзката, а не са останали в локалните TCP буфери.

Следните примери модифицират горния код, използвайки поточна комуникация:

```
// Server code
StreamWriter writer = new StreamWriter(new
  NetworkStream(client));
writer.WriteLine("Welcome to my test server");

// Client code

// Connecting to server...

StreamReader reader = new StreamReader(new
  NetworkStream(socket));
Console.WriteLine(reader.ReadLine());
```

В този пример разчитаме, че сървърът ще изпрати поне един текстов ред. Операцията ще блокира до прочитането му.

TCP комуникация с класа Socket – пример

Ще реализираме едно цялостно решение от две прости приложения, представящи сървър и клиент, които комуникират по TCP със средствата на класа **Socket**. Обърнете внимание, че реализацията тук не ни пази от някои специфични за TCP проблеми на предаването на данни, за които ще стане дума след малко.

1. Отваряме VS. NET и създаваме нов конзолен проект за сървърното приложение.

2. Следният код реализира класа **`Server`**. Обърнете внимание, че в цикъла подаваме на метода **`Send(…)`** съхранената в **`recv`** стойност за брой получени байтове. Ако вместо това подавахме например **`data.Length`**, би трябвало всеки път да реинициализираме масива **`data`**, защото е възможно предишното съобщение да е било по-дълго и новото да е припокрило само част от него, а свойството **`Length`** да е останало непроменено.

```
class Server
{
  public static void Main()
  {
     Socket socket = new Socket(AddressFamily.InterNetwork,
       SocketType.Stream, ProtocolType.Tcp);
     IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 2222);
     socket.Bind(ipep);
     socket.Listen(10);
     Console.WriteLine("Waiting for a client...");
     Socket client = socket.Accept();
     IPEndPoint clientep = (IPEndPoint) client.RemoteEndPoint;
     Console.WriteLine("Connected with {0} at port {1}",
       clientep.Address, clientep.Port);

     string welcome = "Welcome to my test server";
     byte[] data = data = Encoding.ASCII.GetBytes(welcome);
     client.Send(data, data.Length, SocketFlags.None);
     while(true)
     {
       int recv = client.Receive(data);
       if (recv == 0)
         break;
       Console.WriteLine(Encoding.ASCII.GetString(data,0,recv));
       client.Send(data, recv, SocketFlags.None);
     }

     Console.WriteLine("Disconnected from {0}",
       clientep.Address);
     client.Close();
     socket.Close();
   }
}
```

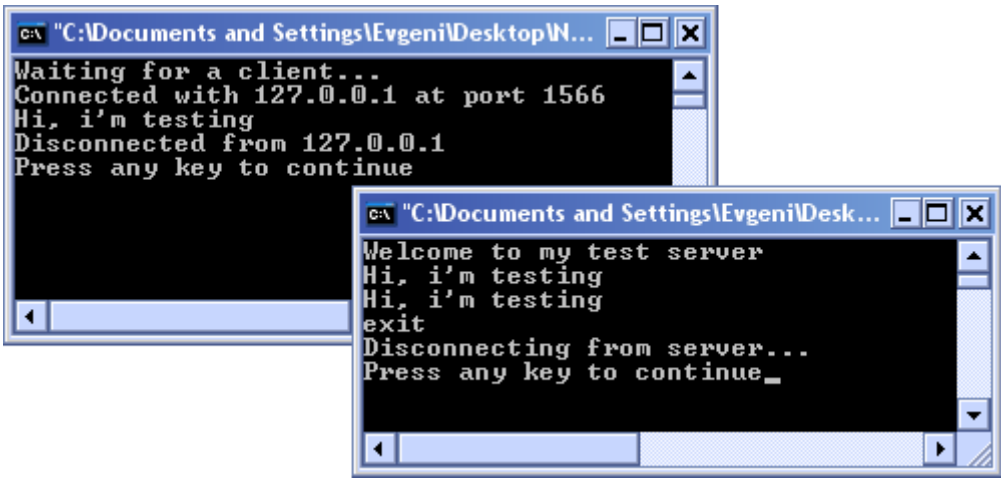
3. Създаваме още един конзолен проект за клиентското приложение. Тук при опита за връзка със сървъра поставяме метода **`Connect(…)`** в **`try-catch`** блок, защото в нормални условия връзката може и да не се осъществи и трябва да реагираме адекватно.

```
class Client
{
```

```
  public static void Main(string[] args)
  {
    Socket socket = new Socket(AddressFamily.InterNetwork,
      SocketType.Stream, ProtocolType.Tcp);
    try
    {
      IPEndPoint ipep = new IPEndPoint(
        IPAddress.Parse("127.0.0.1"), 2222);
      socket.Connect(ipep);
    }
    catch (SocketException e)
    {
      Console.WriteLine("Unable to connect to server.");
      Console.WriteLine(e.ToString());
      return;
    }

    byte[] data = new byte[1024];
    int recv = socket.Receive(data);
    string strData = Encoding.ASCII.GetString(data, 0, recv);
    Console.WriteLine(strData);
    while(true)
    {
      string input = Console.ReadLine();
      if (input == "exit")
        break;
      socket.Send(Encoding.ASCII.GetBytes(input));
      data = new byte[1024];
      recv = socket.Receive(data);
      strData = Encoding.ASCII.GetString(data, 0, recv);
      Console.WriteLine(strData);
    }

    Console.WriteLine("Disconnecting from server...");
    socket.Shutdown(SocketShutdown.Both);
    socket.Close();
  }
}
```

4. Стартираме първо сървърното, а после и клиентското приложение. Опитваме да изпратим няколко съобщения, след което прекратяваме връзката от страна на клиента с командата "exit".

   Можем да видим, че ако например затворим насилствено сървъра, ще предизвикаме изключение при клиента.



Проблеми при TCP връзките с класа Socket

Горният пример демонстрира проста реализация на сървър и клиент, използващи TCP връзка. В повечето случаи тя не е достатъчна за осигуряването на надеждна комуникация. TCP има няколко особености, които трябва да се имат предвид при работа с по-общия клас **`Socket`**, защото често се оказват причина за проблеми. Един от проблемите е размерът на буфера, подаван на метода **`Receive(…)`**.

Този буфер обикновено е по-голям от необходимото, при което последователни обръщания към **`Receive(…)`** пълнят различна част от него, без да изтриват старата информация. За да се предпазим от некоректно извеждани съобщения, смесени с части от предишни, винаги трябва или да инициализираме наново буфера, или задължително да извеждаме само толкова байта от него, колкото ни е указал като върната стойност **`Receive(…)`**. Ако буферът е по-малък пък, ще ни трябват повече обръщения към метода за едно съобщение, което води до логически проблеми.

Основното неудобство на TCP протокола е, че той не държи сметка за границите между отделните съобщения, които се изпращат по мрежата. Те минават първо през TCP буферите на операционната система, откъдето след това нашето приложение ги взима чрез метода **`Receive(…)`**. Ако преди да сме поискали всички данни от буфера, пристигне ново съобщение, то влиза в опашката и вече няма начин да се намери границата между недопрочетеното първо и новопростигналото второ съобщение. Допълнителна опасност е, че това рядко се забелязва при тестове с локален сървър и програмистите не усещат веднага, че има проблем.

Няколко са основните варианти за справяне с този проблем:

- Всяка страна чака за отговор, преди да изпрати ново съобщение. Това е добро решение, но не винаги е възможно в логиката на програмата.

- Предават се само съобщения с фиксиран размер и се чете от буфера, докато се запълнят байтовете на този размер. Това решение има очевиден недостатък, че ограничава вида на съобщенията, а и ако те са по-къси от този размер, се хаби излишно мрежов трафик.
- Предават се съобщения с променлива дължина, която се подава като 4-байтова данна от тип **int** в началото на съобщението. За целта се използват методите **BitConvertor.GetBytes(int)** и **BitConvertor.ToInt32(byte[])**. Ако си комуникират машини с различно подреждане на байтовете, се използва и универсалното мрежово подреждане чрез **IPAddress.HostToNetworkOrder()** и **IPAddress.NetworkToHostOrder()**. Методът е може би най-доброто решение на проблема.
- Едно друго решение е да се слага предварително указан байт като "маркер за край" след всяко съобщение. След това се чете до този байт. Това налага допълнителна обработка и парсване, а и трябва да сме сигурни, че този байт не се среща в самото съобщение, което е проблем. Методът е най-добър при текстови съобщения и за такива се реализира автоматично в поточната комуникация чрез **StreamWriter** и **StreamReader**, като за маркер се използва символът за нов ред.

Свойства на сокетите и задаване на опции

Класът **Socket** притежава няколко свойства, които ни дават информация за състоянието на сокета. Свойството **Available** например ни връща количеството данни, които в момента могат да бъдат прочетени от сокета. Свойствата **Connected**, **LocalEndPoint** и **RemoteEndPoint** ни дават информация дали, с кого и на кой порт е свързан сокетът, като **RemoteEndPoint** може да се използва само ако сме изпълнили **Bind(…)**. Свойството **Blocking** позволява да създаваме сокети, за които операциите по получаване на съобщения не блокират изпълнението на програмата. За това ще говорим малко по-подробно в следващите редове.

Съществуват и редица опции, които контролират поведението на сокета и могат да се задават в хода на програмата. Това става чрез метода **SetSocketOption(SocketOptionLevel level, SocketOptionName name, int value)**. В зависимост от вида на опцията, методът има варианти, при които последният параметър е **byte[]** и дори по-общото **Object**.

Опциите са разделени на нива, които се съдържат в изброения тип **SocketOptionLevel**. Възможните стойности са **Tcp**, **Udp**, **IP** и **Socket**. Те указват вида на опцията. На всяко ниво има много стойности, съдържащи се в изброения тип **SocketOptionName**, като подавайки някоя от тях на метода **SetSocketOption(…)**, трябва да подадем и съответното й ниво като първи параметър. Параметърът value указва стойността, която искаме да дадем на опцията. Например опцията **ReceiveTimeout** е от ниво **Socket** и я задаваме по следния начин – **SetSocketOption(SocketOptionLevel.**

Socket, SocketOptionName.ReceiveTimeout, 3000). Това указва, че методите за получаване трябва да чакат пакет по 3 секунди, преди да предизвикат изключение. Други интересни опции са **NoDelay**, **MaxConnections**, **Debug**, **IPOptions** и др. Повече информация за специфичната опция, която ви е необходима, можете да намерите в MSDN.

Текущата стойност на опция можем да вземем с метода **GetSocketOption(SocketOptionLevel, SocketOptionName)**, който ни връща обект от тип **Object** и трябва да преобразуваме резултата до типа, който очакваме.

Сокет по протокол UDP

Комуникацията по UDP чрез класа **Socket** не се различава особено от тази по TCP и е по-проста. Не трябва да създаваме връзки с **Listen(…)** и **Connect(…)**, достатъчно е само да изпращаме и приемаме съобщения (UDP datagrams) с методите **SendTo(byte[] data, EndPoint remote)** и **ReceiveFrom(byte[] data, ref EndPoint remote)**.

Socket, който комуникира по UDP, създаваме по следния начин:

```
Socket socket = new Socket(AddressFamily.InterNetwork,
   SocketType.Dgram, ProtocolType.Udp);
```

Две UDP приложения могат да контактуват помежду си на произволни портове, но ако искаме едното да играе ролята на UDP сървър, можем да го свържем с определен порт на операционната система чрез метода **Bind(EndPoint ep)**:

```
IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9050);
socket.Bind(ipep);
```

Сега сървърът ще получава съобщения само на този порт.

При метода **SendTo(…)** чрез параметъра **remote** указваме отдалечения сървър, към който ще изпращаме съобщението. Като изключим този детайл, поведението на метода дублира това на метода **Send(…)**, който вече разгледахме. Аналогично на него, можем да използваме и по-разширената форма **SendTo(byte[] data, int size, SocketFlags flags, EndPoint remote)**, за да укажем точно колко байта от буфера изпращаме.

При метода **ReceiveFrom(…)** подаваме един параметър по референция, в който се запазва информацията за отдалечения сървър, от който е дошло съобщението. Това ни позволява после да изпратим съобщение обратно към същия сървър. Обърнете внимание, че за това се налага преобразуване към абстрактния клас **EndPoint**:

```
EndPoint remote = (EndPoint)(new IPEndPoint(IPAddress.Any, 0));
byte[] data = new byte[1024];
int recv = socket.ReceiveFrom(data, ref remote);
Console.WriteLine("Message received from {0}:",
  remote.ToString());
Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));
string welcome = "Welcome to the .NET course!";
data = Encoding.ASCII.GetBytes(welcome);
socket.SendTo(data, data.Length, SocketFlags.None, remote);
```

Ако ще комуникираме с един единствен сървър, няма нужда да използваме **`SendTo(…)`** и **`ReceiveFrom(…)`**. Подобно на клиентския вариант на **`Socket`** при TCP, трябва само да извикаме метода **`Connect(…)`**, с който да укажем този единствен сървър. При TCP този метод изгражда реална връзка между клиента и сървъра. При UDP такава връзка няма, но методът указва на нашия обект, че той ще изпраща данни винаги към този сървър и ще приема datagram пакети само от него:

```
Socket socketToServer = new Socket(AddressFamily.InterNetwork,
  SocketType.Dgram, ProtocolType.Udp);
IPEndPoint ipep = new IPEndPoint(IPAddress.Loopback, 9050);
socketToServer.Connect(ipep);
string welcome = "Hello, are you there?";
byte[] data = Encoding.ASCII.GetBytes(welcome);
socketToServer.Send(data);
int recv = socketToServer.Receive(data);
```

Проблеми при UDP

При комуникацията по UDP проблемът с границите между съобщенията не съществува. UDP не поддържа вътрешни буфери и всяко съобщение се приема наведнъж. Това обаче на свой ред води до други проблеми.

Ако буферът, който използваме в **`ReceiveFrom(…)`**, е твърде малък, за да побере цялото съобщение, данните на практика ще се изгубят. В тази ситуация се предизвиква изключение от тип **`SocketException`**, което информира, че в буфера не е имало достатъчно място за целия datagram пакет. Ние можем да уловим това изключение и да изпратим съобщение, че има проблем. Това не решава проблема със загубените данни, но все пак дава възможност на другата страна да разбере, че нещо не е наред. Единият вариант за изход от ситуацията е да има протокол между страните за максимална дължина на съобщение. Ако това ограничава логиката ни обаче, можем просто да поддържаме променлива дължина на буфера и когато възникне проблем, да я увеличим и да го реинициализираме, като изпратим съобщение с молба за повторно изпращане. Добре е да се отбележи, че този проблем не съществува при **`UdpClient`** класа, където буферът се увеличава автоматично.

Другият проблем с UDP е възможната загуба на пакети. При TCP системата на протокола се грижи това да не става, но тук нямаме гаранция, че едно изпратено съобщение наистина е пристигнало. Понякога това не е проблем за нашето приложение, но ако държим всяко съобщение да пристига, можем да договорим с другата страна изпращане на потвърждаващ отговор. Тук трябва да се има предвид, че чакането за отговор с **`ReceiveFrom(…)`** и **`Receive(…)`** по принцип е блокираща операция и ако има дълготраен проблем с получаване на пакети, нашето приложение на практика ще остане висящо в безкрайно чакане. За да се справим с това, можем да използваме асинхронни сокети (за които ще стане дума след малко) и обект от класа **`Timer`**, който да следи изминалото време за чакане и да го прекратява, когато то стане много. Алтернативно, можем да използваме задаването на опцията **`ReceiveTimeout`** на сокета, която указва колко време да чакат методите за получаване на съобщения. Когато това време изтече, отново се хвърля **`SocketException`**, който можем да уловим и след като опитаме да изпратим още няколко пъти съобщението, и да уведомим клиента, че има проблем с връзката до сървъра.

Понеже трябва да проверяваме за два типа **`SocketException`** и да обработваме две различни ситуации, в **`catch`** блока е хубаво да проверим вътрешния **WinSock** код на грешка на изключението чрез свойството **`ErrorCode`**. Ако кодът е 10040, проблемът е в малкия буфер. Ако кодът е 10054, проблемът е във връзката до сървъра.

UDP комуникация с класа Socket – пример

С този пример ще покажем едно клиентско и едно сървърно приложение, които демонстрират правилното осигуряване на UDP комуникация чрез класа **`Socket`**, така че да избегнем проблемите, описани по-горе.

Класът **`Server`**, който ще напишем, е обикновен клас, който реализира стратегията за обмен на съобщения, която вече разгледахме. Сървърът изчаква получаване на datagram пакет от някой клиент, след което му изпраща поздравително съобщение и започва да се държи като echo сървър, т.е. изпраща обратно на клиента всяко съобщение, което получи.

По-интересен е класът **`Client`**. В него реализираме отделен метод **`SendReceive(Socket, byte[], IPEndPoint)`**, чрез който ще се възползваме от обяснените по-горе техники за преодоляване на проблемите със загубени пакети и малък буфер. В тялото на метода опитваме да изпратим посоченото във втория аргумент съобщение към сървъра, посочен с адреса си в третия аргумент. Ако при този опит уловим **`SocketException`**, проверяваме типа му и реагираме по съответния начин.

В тялото на клиентската програма изпращаме едно начално съобщение към сървъра на локалния мрежов интерфейс и на порт 2222 (на който слушаме със сървърната програма), след което започваме да четем от конзолата съобщения и да ги изпращаме, чакайки за отговор, като

уведомяваме потребителя, ако и след опитите за повторно изпращане няма отговор. Командата "exit" прекратява работата на клиента.

Ще обърнем внимание, че трябва да осигурим по някакъв начин известие, че клиентът е прекратил връзката, когато това стане. За разлика от TCP, тук сървърът няма как да разбере това автоматично (защото на практика няма връзка) и трябва да го уведомим (например да препратим командата "exit" и към него), за да може той да излезе от цикъла, в който чака съобщения от клиента.

Допълнителни методи за идентификация на клиенти трябва да се разработят, когато сървърът ще работи с повече от един клиент едновременно. Понеже методът **`ReceiveFrom(…)`** приема всички съобщения на UDP порта, на който "слуша" сървъра, винаги трябва да проверяваме адреса, от който идва съобщението, за да знаем от кой клиент идва.

За да построим приложението трябва да изпълним следните стъпки:

1. Отваряме VS .NET и създаваме ново конзолно приложение.
2. Въвеждаме кода на сървърния клас:

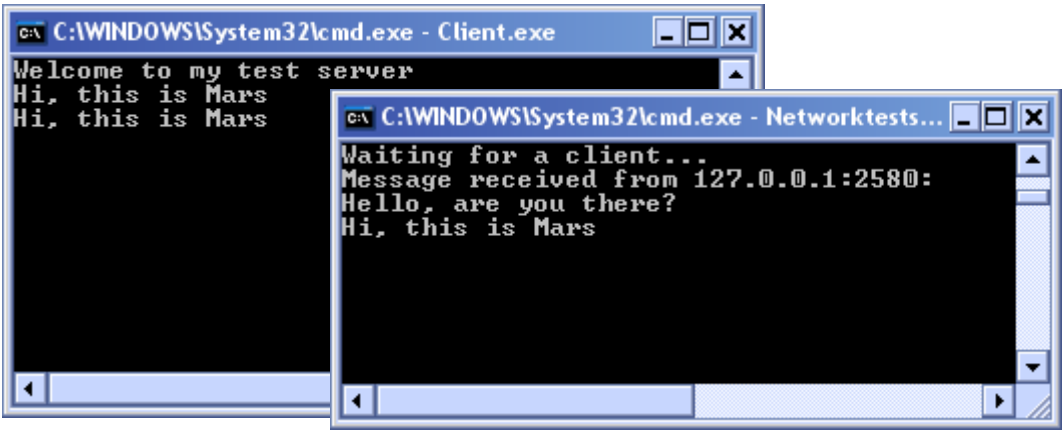
```
class Server
{
  static void Main(string[] args)
  {
    Socket server = new Socket(AddressFamily.InterNetwork,
      SocketType.Dgram, ProtocolType.Udp);
    IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 2222);
    server.Bind(ipep);
    Console.WriteLine("Waiting for a client...");
    IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);
    EndPoint client = (EndPoint)(sender);
    byte[] data = new byte[1024];
    int recv = server.ReceiveFrom(data, ref client);
    Console.WriteLine("Message received from {0}:",
      client.ToString());
    Console.WriteLine(Encoding.ASCII.GetString(data, 0, recv));
    string welcome = "Welcome to my test server";
    data = Encoding.ASCII.GetBytes(welcome);
    server.SendTo(data, data.Length, SocketFlags.None, client);
    while(true)
    {
      data = new byte[1024];
      recv = server.ReceiveFrom(data, ref client);
      Console.WriteLine(
        Encoding.ASCII.GetString(data, 0, recv));
      server.SendTo(data, recv, SocketFlags.None, client);
    }
  }
}
```

3. Създаваме още един конзолен проект и въвеждаме кода на клиентския клас.

```
class Client
{
  const int MAX_RETRY_ATTEMPTS = 4;
  const int BUFFER_INCREMENT = 10;

  private static byte[] data = new byte[1024];
  private static EndPoint remote = (EndPoint)
    new IPEndPoint(IPAddress.Any, 0);
  private static int size = 30;

  private static int SendReceive(Socket s, byte[] message,
    EndPoint server)
  {
   int recv = 0;
   int retry = 0;
   while (true)
   {
     if (retry != 0)
       Console.WriteLine("Retry #{0}", retry);
     try
     {
       s.SendTo(message, message.Length, SocketFlags.None,
         server);

       recv = s.ReceiveFrom(data, ref remote);
     }
     catch (SocketException e)
     {
       if (e.ErrorCode == 10054)
         Console.WriteLine("Error connecting to server");
       else if (e.ErrorCode == 10040)
       {
         Console.WriteLine("Error receiving packet");
         size += BUFFER_INCREMENT;
         data = new byte[size];
       }
       recv = 0;
     }
     if (recv > 0) return recv;
     else
     {
       retry++;
       if (retry > MAX_RETRY_ATTEMPTS) return 0;
     }
    }
  }

  public static void Main()
```

```
  {
   IPEndPoint ipep = new IPEndPoint(
     IPAddress.Parse("127.0.0.1"), 2222);
   Socket client = new Socket(AddressFamily.InterNetwork,
     SocketType.Dgram, ProtocolType.Udp);
   client.SetSocketOption(SocketOptionLevel.Socket,
     SocketOptionName.ReceiveTimeout, 3000);
   string greeting = "Hello, are you there?";
   data = Encoding.ASCII.GetBytes(greeting);
   int recv = SendReceive(client, data, ipep);
   if (recv > 0)
   {
     string strData = Encoding.ASCII.GetString(data, 0, recv);
     Console.WriteLine(strData);
   }
   else
   {
     Console.WriteLine("Unable to talk with remote host");
     return;
   }
   while(true)
   {
     string input = Console.ReadLine();
     if (input == "exit")
       break;
     recv = SendReceive(
       client, Encoding.ASCII.GetBytes(input), ipep);
     if (recv > 0)
     {
       strData = Encoding.ASCII.GetString(data, 0, recv);
       Console.WriteLine(strData);
     }
     else
       Console.WriteLine("Did not receive an answer");
   }
   Console.WriteLine("Stopping client");
   client.Close();
  }
}
```

4. Стартираме сървъра и го оставяме да слуша за идващи съобщения. След това стартираме и клиента. Примерен резултат след няколко съобщения изглежда по следния начин:



Няколко думи за асинхронните сокети

Както вече изяснихме, всички сокети блокират на определени операции като например **`Accept()`**, **`Connect(…)`**, **`Send(…)`** и **`Receive(…)`**, докато тези методи не завършат действието си. Понеже това може да продължи безкрайно, в много случаи е добре по някакъв начин да можем да възобновим хода на програмата след блокиращата операция, дори тя да е в крайна сметка неуспешна. Това важи в особено голяма степен за графични приложения, където не е правилно да позволяваме цялото приложение да отнема контрола на потребителя докато чака блокиращ метод, който може да не завърши по ред причини, например стабилността на мрежата.

Свойството Blocking

За да се справим с този проблем, можем да използваме свойството **`Blocking`**, за което вече стана дума. Ако зададем на това свойство стойност **`false`**, то сокетът престава да третира тези операции като блокиращи. Това означава, че например методът **`Receive(…)`** ще провери дали има данни за получаване и ако да – ще ги получи, – а ако няма, ще върне 0 и ще завърши, без да чака да се появят някакви данни (както е по подразбиране).

Асинхронни методи

По-добър вариант (доколкото ни дава повече възможности за обработка) е да използваме т. нар. асинхронни методи на класа **`Socket`**. Това са двойки методи от типа **`BeginXXX(…)`** и **`EndXXX(…)`**, които отговарят на стандартните методи и имплементират стандартния за .NET модел на асинхронно извикване на методи.

Ще дадем пример с метода **`Accept()`**. Както знаем, той се извиква при сървъра и блокира, докато не се появи клиентска връзка, като връща в резултат новосъздаден сокет за тази връзка. Ако искаме потребителят да е в състояние да извършва и други задачи, докато сървърът чака за клиентско запитване, използваме метода **`void BeginAccept(`**

AsyncCallback callback, Object state). Първият параметър е делегат, който сочи към функцията, която ще се изпълни, когато се появи клиентска връзка. В обработката на тази функция трябва да извикаме метода **Socket EndAccept(IAsyncResult result)**, който вече ще ни даде като резултат новия сокет.

Понеже делегатът **callback** е със зададен прототип, не можем да му подадем като параметър самия сокет, за който трябва да извикаме **EndAccept(…)**. По тази причина сокета подаваме като втори аргумент на **BeginAccept(…)**. Този аргумент е от тип **Object** и служи именно за предаване на необходима информация между основната програма и callback метода, където можем да използваме сокета, извличайки го от свойството **AsyncState** на параметъра **result**:

```
static void Main(string[] args)
{
   Socket socket = new Socket(AddressFamily.InterNetwork,
     SocketType.Stream, ProtocolType.Tcp);
   IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 2222);
   socket.Bind(ipep);
   socket.Listen(10);
   socket.BeginAccept(new AsyncCallback(AcceptCallback), socket);
   // Do some stuff
}

private static void AcceptCallback(IAsyncResult iar)
{
   Socket socket = (Socket)iar.AsyncState;
   Socket client = socket.EndAccept(iar);
   // Send message to client, handle communication
}
```

По същия начин процедираме с методите **BeginSend(…)**, **EndSend(…)**, **BeginConnect(…)**, **EndConnect(…)** и т. н. Ще отбележим само, че тези методи са на разположение само за класа **Socket**. По-специализираните класове, които разгледахме (**UdpClient**, **TcpClient** и **TcpListener**) не поддържат асинхронно извикване.

Методите Poll(...) и Select(...)

Класът предлага още два метода, които можем да използваме, за да не позволим на сокета да остане в блокирано състояние.

Проверка за блокиране с Poll(...)

Методът **Poll(int time, SelectMode mode)** проверява в продължение на **time** микросекунди (1 секунда = 1000000 микросекунди) дали сокетът ще блокира на дадена операция, определена от параметъра **mode**. Ако това е

така, **Poll(…)** връща **false** и ние знаем да не викаме метода, който би блокирал изпълнението на програмата.

SelectMode е изброен тип и съдържа три стойности: **SelectRead**, **SelectWrite** и **SelectError**. Първата проверява дали има връзки за приемане и дали има данни за получаване и връща true, ако някое от тях е вярно или ако връзката е затворена. **SelectWrite** връща **true**, ако сокетът е бил свързан с **Connect(…)** или ако по връзката могат да се изпратят данни, а **SelectError** проверява дали е имало грешки при **Connect(…)** и предаването на данни.

Следният пример показва как да използваме **Poll(…)**, за да четем данни в неблокиращ режим при създадени променливи **socket** (за сървърната страна) и **client** (за клиентската страна). Отново реализираме познатия сървър, който повтаря получените от клиента съобщения.

```
while(true)
{
   result = client.Poll(3000000, SelectMode.SelectRead);
   // Do some stuff
   if(result)
   {
      data = new byte[1024];
      recv = client.Receive(data);
      if (recv == 0)
        break;
      Console.WriteLine(
        Encoding.ASCII.GetString(data, 0, recv));
      client.Send(data, recv, 0);
   }
   else
   {
    // Do other stuff
   }
}
```

Възможности на метода Select(...)

Статичният метод **Select(IList checkRead, IList checkWrite, IList checkError)** извършва същите действия, но за повече обекти от класа **Socket** едновременно. Всички такива обекти, съдържащи се в колекцията **checkRead**, се проверяват със **SelectRead**; всички в **checkWrite** – със **SelectWrite** и т.н. След изпълнението на метода колекциите съдържат само тези сокети, за които върнатата стойност (при проверка като с **Poll(…)**) е true. Този метод е полезен, когато работим с няколко клиента едновременно – тогава можем да поставим техните сокети в колекциите и да ги проверим с метода **Select(…)**. След завършването на метода, обхождаме колекциите и извършваме съответните действия, за които вече знаем, че няма да блокират изпълнението. Не бива да се забравя, че

Select(…) променя подадените му колекции. Ако искаме да запазим всички сокети, трябва да ги пазим в друга колекция при извикването на метода.

Multicasting в .NET Framework

Понякога е удобно дадено съобщение да се изпраща на всички клиенти в локалната мрежа, например за реклама. Това става, като изпратим пакет на специален запазен адрес, и се нарича **broadcasting**. **Multicasting** се нарича изпращането на пакет на група от предварително зададени адреси наведнъж. Тук ще разгледаме накратко средствата в .NET Framework за реализиране на broadcasting с multicasting.

Broadcasting сокети

В описанието на класа **IPAddress** споменахме полето **IPAddress.Broadcast**, което ни дава локалния broadcast адрес. Ако изпратим съобщение на този адрес чрез метода **SendTo(…)**, следва то да се получи от всички свързани в локалната мрежа абонати. Понеже .NET няма да ни позволи да направим това с обикновен, създаден по подразбиране, сокет, трябва да променим стойността на опцията **Broadcast** по следния начин:

```
Socket broadcast = new Socket(AddressFamily.InterNetwork,
   SocketType.Dgram, ProtocolType.Udp);
broadcast.SetSocketOption(SocketOptionLevel.Socket,
   SocketOptionName.Broadcast, 1);
string message = "This is a broadcast message";
byte[] data = Encoding.ASCII.GetBytes(message);
broadcast.SendTo(data,
  new IPEndPoint(IPAddress.Broadcast, 2222));
```

Получаването на broadcast съобщения от страна на клиента не се различава от обикновено получаване на съобщения с **ReceiveFrom(…)**. Особеност на broadcast изпращането е единствено, че сокетът трябва да работи с datagram пакети, т.е. трябва да използваме **SocketType.Dgram** за тип на сокета и обикновено работим с UDP.

Multicasting сокети

За разлика от broadcasting, multicasting комуникацията не е ограничена само в рамките на локалната мрежа. За multicasting са резервирани всички IP адреси в глобалната мрежа в интервала 224.0.0.1 до 239.255.255.255. Всеки от тези адреси представя една т.нар. multicast група. Едно приложение може да се абонира за подобна група, при което то ще получава всички пакети, изпратени към IP адреса на групата.

За създаването на сокет, който да комуникира с multicast съобщения, отново трябва да зададем определена опция. Абонирането за дадена група можем да видим в примера:

```
Socket multicast = new Socket(AddressFamily.InterNetwork,
   SocketType.Dgram, ProtocolType.Udp);
IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 2222);
multicast.Bind(ipep);
multicast.SetSocketOption(SocketOptionLevel.IP,
  SocketOptionName.AddMembership,
  new MulticastOption(IPAddress.Parse("224.0.0.56")));
```

Използваме опцията **AddMembership**, която приема за стойност обект от тип **MulticastOption**, чийто конструктор на свой ред приема като стойност **IPAddress** и тук подаваме валиден multicast групов адрес. Обърнете внимание, че метода **SetSocketOption(…)** трябва да извикаме след метода **Bind(…)**. Оттук нататък сокетът **multicast** ще получава както съобщения, определени за адреса, за който е извикан **Bind(…)**, така и такива, определени за адреса на multicast групата.

За изпращане на multicast съобщения не се нуждаем от специална настройка на опциите, а просто изпращаме съобщение на съответния multicast адрес.

Една особеност на този вид комуникация е т.нар. Time To Live (TTL) стойност на IP пакетите. Тя по подразбиране е 1, което значи, че нашите пакети не могат да преминат през маршрутизатор (router) и на практика multicast съобщенията отново се предават само в рамките на локалната мрежа, ако адресът е там. TTL времето можем да подобрим със следното задаване на опцията **MulticastTimeToLive**:

```
multicast.SetSocketOption(SocketOptionLevel.IP,
  SocketOptionName.MulticastTimeToLive, 50);
```

Ще отбележим само още, че група може да бъде и напусната като зададем multicast адреса, с който сме работили, при промяна стойността на опцията **DropMembership**:

```
multicast.SetSocketOption(SocketOptionLevel.IP,
   SocketOptionName.DropMembership,
   new MulticastOption(IPAddress.Parse("224.0.0.56")));
```

Multicasting с класа UdpClient

Класът **UdpClient** също поддържа методи за multicasting. Това са **JoinMulticastGroup(IPAddress, int)** и **LeaveMulticastGroup (IPAddress, int)**. Тук първият параметър указва адреса, който представя multicast групата, а вторият задава директно TTL стойността.

Изпращането и приемането на съобщения не се различава особено от вече разгледаното при **Socket**. Единствено трябва да помним да подаваме на конструктора на **UdpClient** номера на порта, на който искаме да

получаваме съобщения, защото ако няма зададен порт, **`JoinMulticastOption(…)`** ще се провали и ще предизвика изключение.

Използване на DNS услуги чрез класа Dns

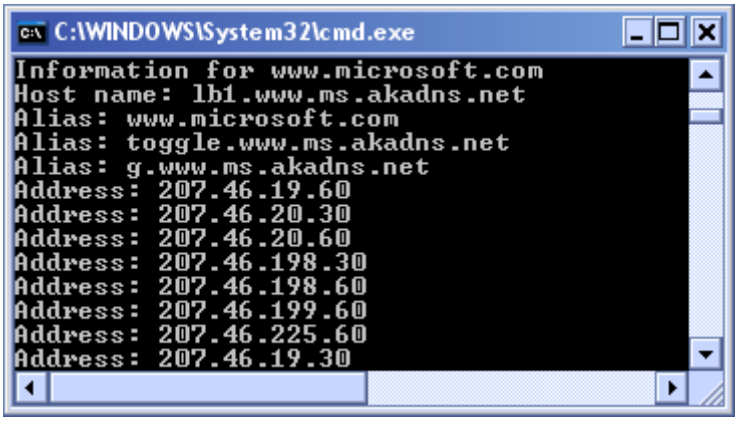
За изпълняване на DNS заявки в .NET Framework използваме класа **`Dns`**. С помощта на неговите методи можем да проверим имената на домейните, които отговарят на дадени адреси на машини, както и обратното.

Резултатът от DNS заявките, изпълнявани чрез класа **`Dns`**, е във вид на обекти от класа **`IPHostEntry`**. Този клас има няколко свойства, в които се запазва информацията, която ни интересува. Името на домейна се намира в свойството **`HostName`**. В списъка от **`IPAddress`** обекти **`AddressList`** се пазят всички адреси, асоциирани с този домейн, а в списъка от низове **`Aliases`** – всички псевдоними на домейна, чрез които той също е достъпен.

Основният метод, който ще използваме, е методът **`Resolve(string)`**. Той връща обект от типа **`IPHostEntry`**, а приема за параметър един низ, който може да е или име на домейн, или IP адрес (в стандартния вид с четири еднобайтови числа, разделени с точки). Следният пример демонстрира използването му:

```
IPHostEntry iphe = Dns.Resolve("www.microsoft.com");
Console.WriteLine("Information for www.microsoft.com");
Console.WriteLine("Host name: {0}", iphe.HostName);
foreach(string alias in iphe.Aliases)
{
   Console.WriteLine("Alias: {0}", alias);
}
foreach(IPAddress address in iphe.AddressList)
{
   Console.WriteLine("Address: {0}", address.ToString());
}
```

Ето и резултата – имената и псевдонимите на хоста www.microsoft.com:



Освен метода **Resolve(…)**, имаме на разположение и методите **GetHostByName(…)** и **GetHostByAddress(…)**, които приемат съответно име на домейн и адрес, но тъй като понякога не се знае по какво ще искаме да търсим (например при вход от потребителя, който може да въведе както IP адрес, така и име на домейн), е за препоръчване да използваме метода **Resolve(…)**.

Асинхронни DNS заявки

Класът **Dns** предлага и асинхронни заявки, например чрез методите **BeginResolve(string, AsyncCallback, object)** и **EndResolve(AsyncResult)**. Ако не искаме програмата ни да блокира при евентуално чакане за отговор от DNS сървър, можем да ги използваме в една примерна преработка на горния пример така:

```
static void Main(string[] args)
{
   Object state = new Object();
   AsyncCallback OnResolved = new AsyncCallback(Resolved);
   Dns.BeginResolve("www.microsoft.com", OnResolved, state);
   // Do some other stuff
   Console.Read(); // This is to prevent program termination
}

private static void Resolved(IAsyncResult ar)
{
   IPHostEntry iphe = Dns.EndResolve(ar);
   Console.WriteLine("Host name: {0}",iphe.HostName);
   foreach(string alias in iphe.Aliases)
   {
      Console.WriteLine("Alias: {0}",alias);
   }
   foreach(IPAddress address in iphe.AddressList)
   {
      Console.WriteLine("Address: {0}",address);
   }
}
```

Използваме метода **Console.Read()** за да забавим изпълнението на програмата – в противен случай тя ще приключи веднага след извикването на **BeginResolve(…)** и няма да успеем да видим ефекта от последващото асинхронно извикване на метода **Resolved(…)**. Въпросния метод правим статичен, за да можем да го използваме при създаването на делегата **OnResolved** в статичния контекст на **Main(…)**.

Работа с уеб ресурси – класът WebClient

Именното пространство **System.Net** ни предоставя и удобния клас **WebClient**, чрез който можем лесно да извършваме проста комуникация

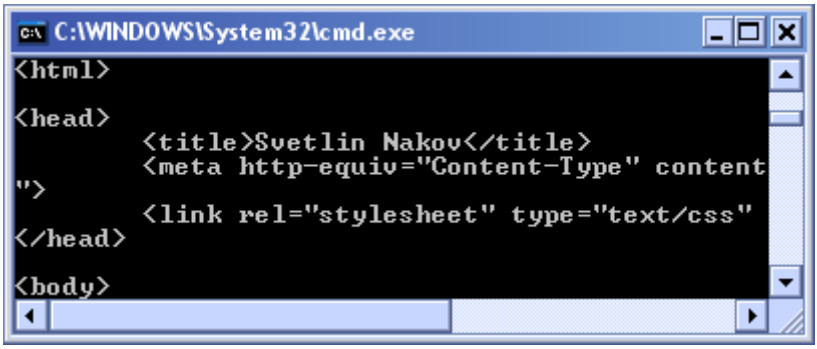
по HTTP протокола с някой уеб-сървър. Методите на класа са съсредоточени в две основни направления – за извличане (download) на данни и за изпращане (upload) на данни.

Извличане на данни по HTTP

Класът **WebClient** ни позволява да получим резултата от HTTP заявка във вид на HTML (т.нар. **raw HTML**), запазен в масив от тип **byte**. За целта използваме метода **DownloadData(string URI)**. Като аргумент подаваме адреса на уеб-ресурса (обикновено адрес на Интернет страница), който искаме да изтеглим. Това е низ, образуван по стандартните правила за URI. Можем да извличаме няколко типа ресурси – HTTP (**http://**), HTTP по SSL канал (**https://**), както и локални ресурси (**file://**). Следният пример демонстрира употребата на метода **DownloadData(…)**:

```
static void Main(string[] args)
{
   WebClient wc = new WebClient();
   byte[] response = wc.DownloadData("http://www.nakov.com");
   Console.WriteLine(Encoding.ASCII.GetString(response));
}
```

В резултат получаваме HTML съдържанието на посочената страница:



Както обикновено, текстовото представяне на получените данни получаваме чрез методите на класа **Encoding**. Обърнете още веднъж внимание, че тук трябва да се подаде пълен URI низ за ресурс. Това е разлика например с класа **Dns** и класа **TcpClient**, където подавахме само име на домейн, и трябва да се внимава за грешки. Ако не сложим "http://" отпред, **WebClient** ще се опита да търси ресурса "www.nakov.com" в локалната файлова система и ще се предизвика изключение от тип **WebException**. Тези изключения описват различните проблеми, които методите за HTTP комуникация могат да срещнат при изпълнението си.

Следващата схема илюстрира модела "заявка-отговор", който се използва при извличане на уеб ресурси по протокол HTTP:

Изтегляне на файл от URL

Ако искаме да изтеглим файл от даден адрес, можем да го направим отново с метода **DownloadData(…)** и да обработим получения масив от тип **byte[]**. По-лесно обаче е да използваме предоставения метод **DownloadFile(string URI, string filename)**. При него като втори аргумент подаваме име на файл на локалната файлова система и .NET Framework се грижи да запише в него извлечените данни:

```
static void Main()
{
   WebClient wc = new WebClient();
   wc.DownloadFile("http://www.nakov.com", "data.html";);
   Console.WriteLine("File downloaded");
}
```

Ако сега проверим съдържанието на файла **data.html**, който трябва да е запазен в папката **bin\Debug** на нашето приложение (където е стартиран и изпълнимият файл на програмата), ще видим, че то е същото като изхода от първия пример.

Извличане на данни от URL чрез поток

Класът **WebClient** предлага още един метод за download – това е методът **OpenRead(string URI)**. Подобно на **DownloadData(…)** той извлича raw HTML от отговора на HTTP заявката към посочения от URI ресурс, но я предоставя във вид на поточен обект от класа **Stream**. Това ни позволява да обработваме информацията на порции, а не наведнъж. Създавайки един четец от типа **StreamReader** на базата на получения поток, можем да получим по-гъвкави възможности за обработка. Следващият пример показва как да получим информацията от сървъра в поточен вид с **OpenRead(…)**:

```
WebClient client = new WebClient();
Stream stream = client.OpenRead("http://www.nakov.com");
StreamReader reader = new StreamReader(stream);
while (true)
{
   string response = reader.ReadLine();
   if (response == null) {
      break;
   }
   Console.WriteLine(response);
}
reader.Close();
```

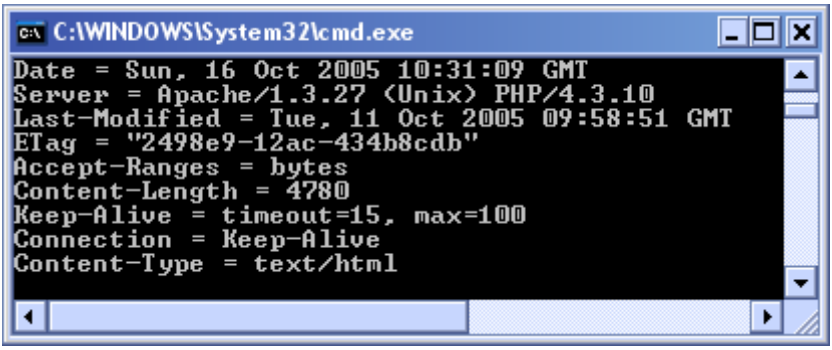
Този код извежда същия резултат като преобразуването на масива от байтове от **DownloadData(…)** в низ, но не го прави наведнъж, а последователно, и ни позволява, ако желаем, да извършваме някаква обработка на данните, докато ги четем.

Проверка на HTTP хедърите

Във всеки HTTP отговор освен данните на самия поискан ресурс се включват и различни части помощна информация, които наричаме **HTTP headers**. Това са например данни за кодовата таблица на извличания текст, за характеристиките на уеб-сървъра, за кеширането на уеб-ресурса и т.н. Те не се включват в raw HTML частта и не можем да ги видим чрез **DownloadData(…)** но свойството **ResponseHeaders** на класа **WebClient** ни позволява след извършена заявка да проверим стойностите на тези редове с метаданни. Свойството **ResponseHeaders** по същество е една колекция от тип **WebHeaderCollection**, която обхождаме по стандартния начин, извличайки ключовете и прилежащите им стойности, например така:

```
WebClient client = new WebClient();
byte[] data = client.DownloadData("http://www.nakov.com");

WebHeaderCollection headers = client.ResponseHeaders;
for (int i=0; i<headers.Count; i++)
{
   string key = headers.Keys[i];
   string val = headers[i];
   Console.WriteLine("{0} = {1}", key, val);
}
```

Резултатът от изпълнението на този код е следният:



Изпращане на данни по HTTP

Изпращането на данни към уеб сървър става по почти аналогичен начин. Разполагаме с методите **`UploadData(…)`** и **`UploadFile(…)`**, които ни позволяват да изпратим обобщени данни или цял файл по връзката със сървъра. Разбира се, в общия случай трябва да имаме съответните права, за да го направим. Как да укажем парола и потребителско име, за да получим тези права, ще обясним след малко.

Методът **`UploadData(string URI, string method, byte[] data)`** изпраща масив от байтове към сървъра. Употребата му е подобна на **`DownloadData(…)`**, но като втори параметър можем да укажем (незадължително) метода на HTTP заявката. По подразбиране той е "**`POST`**", но можем да използваме и "**`GET`**".

Методът **`UploadFile(string URI, string method, string filename)`** ни позволява upload на локален файл (указан от **`filename`**) на сървъра. И двата метода връщат като резултат масив от тип **`byte[]`**, в който се съдържа евентуален HTTP отговор от страна на сървъра. Следният пример демонстрира използването им:

```
WebClient wc = new WebClient();
string data = "This is the data to post";
byte[] array = Encoding.ASCII.GetBytes(data);
wc.UploadData(args[0], array);
wc.UploadFile(args[0], "file.zip");
```

С този код ще изпращаме първо един низ, а после и файла "file.zip" (който, разбира се, трябва да съществува) на адреса, посочен като параметър на командния ред на програмата.

Подобно на **`OpenRead(…)`**, **`WebClient`** предлага и метод за поточно изпращане на данни към сървъра. Неговата сигнатура е **`OpenWrite(string URI, string method)`**, отново с незадължителен втори аргумент, и в резултат от изпълнението му получаваме обект от тип **`Stream`**, чрез който можем да създадем по-удобния за употреба **`StreamWriter`**.

Изпращане на параметри към HTTP заявка

Методът **UploadValues(string URI, string method, NameValueCollection values)** е малко по-различен от горните три. Той не се използва за upload към сървъра, а служи за подаване на параметри на обработващ скрипт, симулирайки изпращането им чрез HTML форма. Ако методът на изпращането е "**GET**", те се добавят към URI адреса във формата **?name=value&name=value** и т.н.; а ако методът е "**POST**", се изброяват в тялото на HTTP заявката.

Примерът, който ще дадем, използва поддържаната от локален IIS сървър страница "**testform.aspx**", но по желание може да се използва произволен адрес, за който се очаква да обработва резултати от форма. Както и при **UploadData(…)** и **UploadFile(…)** резултатът от метода е реално HTTP отговорът от страна на сървъра във вид на **byte[]** масив. Чрез него можем да разберем как е реагирала програмата за обработка на данните, които подаваме.

```
string uri = "http://localhost/testform.aspx";
NameValueCollection nvcn = new NameValueCollection();
nvc.Add("lastname", "Dijkstra");
nvc.Add("firstname", "Edsgar");
WebClient wc = new WebClient();
byte[] response = wc.UploadValues(uri, nvc);
Console.WriteLine(Encoding.ASCII.GetString(response));
```

Автентикация с Credentials

Често при връзка с различни уеб-сървъри се налага да се автентикираме пред тях, за да получим необходимите права за извличане и изпращане на различни уеб-ресурси. Това можем да направим, използвайки свойството **Credentials** на класа **WebClient**.

Свойството **Credentials** поддържа стойности от два класа в пространството **System.Net** – **NetworkCredential** и **CredentialCache**. Обектите от класа **NetworkCredential** представят една комбинация от потребителско име и парола (а за Windows сървъри – и име на домейн). Такъв обект можем да създадем с конструктора **NetworkCredential(string username, string password)** или чрез конструктора **NetworkCredential()** и последващо установяване на свойствата **UserName** и **Password**. Присвоявайки на **Credentials** така създадения обект вече можем да се свържем със сайт, който изисква автентикация, например чрез **DownloadData(…)**. Ако името и паролата са верни, ще получим резултата от HTTP отговора, в противен случай ще се предизвика изключение:

```
WebClient wc = new WebClient();
string uri = "http://localhost/testlogin";
NetworkCredential nc = new NetworkCredential("user","pass");
```

```
wc.Credentials = nc;
try
{
   byte[] response = wc.DownloadData(uri);
   Console.WriteLine(Encoding.ASCII.GetString(response));
}
catch (WebException exception)
{
   Console.WriteLine("Try different username/password");
}
```

Освен **NetworkCredential**, можем да използваме и класа **CredentialCache**. Обектите от този клас съхраняват записи за адреси и съответните им обекти от тип **NetworkCredential**. Записи можем да добавяне с метода **Add(string URI, string authmode, NetworkCredential credential)**. При този метод вторият параметър указва типа на автентикация ("Basic" или "Digest" за MD5 хеширане на данните).

Ако присвоим на свойството **Credentials** обект от типа **CredentialCache**, то при всяко извикване на **DownloadData(…)** или останалите методи за връзка със сървър, .NET проверява дали някой от записите не съвпада с търсения адрес и ако има такъв – използва съответния **NetworkCredential** обект за автентикация.

Други полезни свойства на WebClient

Свойството **BaseAddress** ни позволява да зададем общ адрес за всички операции, извършвани с **WebClient**. Този адрес се добавя в началото на указания в съответния аргумент на методите за извличане и изпращане на данни адрес. Това се използва, когато ще се свързваме само с един и същи сайт, от който ни трябват различни ресурси, които можем да зададем по-кратко с относителен път спрямо **BaseAddress**:

```
WebClient wc = new WebClient();
wc.BaseAddress = "http://www.devbg.org/";
byte[] response = wc.DownloadData("dotnetbook");
Console.WriteLine(Encoding.ASCII.GetString(response));
```

Друго интересно свойство е **QueryString**, с което можем да извършим нещо подобно на метода **UploadValues(…)**, използвайки **GET**. **QueryString** приема за стойност също **NameValueCollection** от имена на параметри и техните стойности, които после се добавят в стандартния формат на края на подаденото URI при всяка заявка за ресурс към уеб-сървър.

HTTP заявки с класовете HttpWebRequest и HttpWebResponse

Класът **WebClient** ни предоставя основната функционалност, която обикновено е нужна за работа с HTTP протокола. Ако искаме по-фин контрол върху заявките ни и имаме намерение да обработваме по-гъвкаво резултата от отговора, можем да използваме специализираните класове **HttpWebRequest** и **HttpWebResponse**. За достъп до локални ресурси са подходящи класовете **FileWebRequest** и **FileWebResponse**, но понеже те предоставят същата функционалност като тези за работа с HTTP, няма да ги разглеждаме специално.

Създаване на HHTP заявка

Класът **HttpWebRequest** е наследник на абстрактния клас **WebRequest** и за създаването му използваме метода **Create(string URI)**, като после преобразуваме резултата до желания тип:

```
HttpWebRequest request =
  (HttpWebRequest) WebRequest.Create("http://www.nakov.com");
```

След като сме създали по този начин обекта на заявката, можем да настроим по желание неговите свойства. Класът **HttpWebRequest** предлага голямо количество такива свойства, които да укажат точно особеностите на заявката. Например свойството **Method** указва метода на заявката (**GET**, **POST**, **HEAD**, **PUT** и др.); свойството **AllowAutoRedirect** указва дали заявката автоматично да се подаде отново към посочения адрес, ако в отговор получи указание за пренасочване; а свойства като **Accept**, **ContentType** или **UserAgent** задават стойности на съответните заглавни полета (HTTP headers) на заявката. Всички headers можем да прегледаме чрез свойството **Headers**, което както при **WebClient** има за стойност обект от тип **WebHeaderCollection**. Използвайки това свойство, можем и да създаваме нови заглавни полета (освен предлаганите от класа), като просто ги добавяме към колекцията.

Чрез свойството **Proxy** можем да укажем **proxy server**, през който да минава нашата заявка. Това е полезно за връзка с някои ресурси зад защитни стени (**firewalls**) и др. Стойността на това свойство е от тип **WebProxy** и можем да го зададем по следния начин:

```
WebProxy proxy = new WebProxy("193.95.112.71:8080");
request.Proxy = proxy;
```

Изпращане на данни към HTTP сървър

Данни към посочения в конструктора или в свойството **RequestUri** адрес можем да изпращаме чрез метода **GetRequestStream()**. Като резултат от

извикването му получаваме един обект от класа **Stream**, чрез който можем да подаваме данни, използвайки за удобство и класа **StreamWriter**:

```
HttpWebRequest request = (HttpWebRequest)WebRequest.Create(
  "http://localhost/testform.aspx");
request.Method = "POST";
string postData = "firstname=Edsgar";
request.ContentType="application/x-www-form-urlencoded";
request.ContentLength=postData.Length;
Stream stream = request.GetRequestStream();
StreamWriter sw = new StreamWriter(stream);
sw.Write(postData);
stream.Close();
```

Обърнете внимание, че се налага да сменим метода на **request** от **GET** на **POST**, защото в противен случай **GetRequestStream()** ще предизвика изключение. Това е така, понеже не можем да изпращаме данни с метода **GET**, който по принцип служи за изтегляне на данни. Освен това трябва да укажем количеството на информацията, която изпращаме, задавайки съответната стойност на свойството **ContentLength**.

Не бива да забравяме да затворим потока, когато приключим с него. В противен случай последващи изпълнения на нашата HTTP заявка ще предизвикват изключения, защото връзката не е приключила.

Получаване на HTTP отговор

По-често ние не искаме да изпращаме данни, а просто да извлечем отговор на заявка за определен уеб-ресурс. За целта използваме метода **GetResponse()**. Той връща обект от класа **HttpWebResponse**, който съдържа отговора на сървъра.

От всеки HTTP отговор обикновено ни интересуват няколко неща – съдържанието на заглавните полета, HTTP статус кода и съдържанието на самия отговор.

Първото можем да извлечем по няколко начина. За по-често срещаните заглавни полета, като например **Content-Type**, **Content-Length**, **Content-Encoding** има специални свойства на класа **HttpWebResponse**, които можем да достъпваме, съответно **ContentType**, **ContentLength** и т.н. Можем и да извлечем стойността на произволно заглавно поле (ако такъв има в отговора) чрез метода **GetResponseHeader(string header)**. Ако искаме направо всички заглавни полета, можем да използваме и познатото свойство **Headers**, което отново ни дава колекция от двойки име-стойност.

Свойството **StatusCode** ни дава информация за HTTP статус кода на съобщението от сървъра. Кодовете са много и различни, като различните им поредни цифри образуват смисъла на цялото съобщение. Код 200 означава успешна заявка и последващ отговор, а кодовете с вида 40X се

използват за грешки. Повече информация може да се намери в Интернет, например RFC 2616 (http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html). **StatusCode** е от тип **HttpStatusCode** – изброен тип, чиито стойности отговарят на различните кодове, например **HttpStatusCode.OK** (за код 200) или **HttpStatusCode.NotFound** (за код 404).

Съдържанието на отговора можем да прочетем, създавайки поток за четене от **HttpWebResponse** чрез метода **GetResponseStream()**. Този метод отново ни дава обект от класа **Stream**, който да използваме за четене чрез създаването на **StreamReader**:

```
string url = "http://www.devbg.org/";
HttpWebRequest request =
   (HttpWebRequest) WebRequest.Create(url);
HttpWebResponse response =
   (HttpWebResponse) request.GetResponse();
string contentType = response.ContentType;
Console.WriteLine("Content-Type: {0}", contentType);
Stream stream = response.GetResponseStream();
using (stream)
{
   StreamReader sr = new StreamReader(stream);
   string responseBody = sr.ReadToEnd();
   Console.WriteLine(responseBody);
}
```

Извличане на Cookies

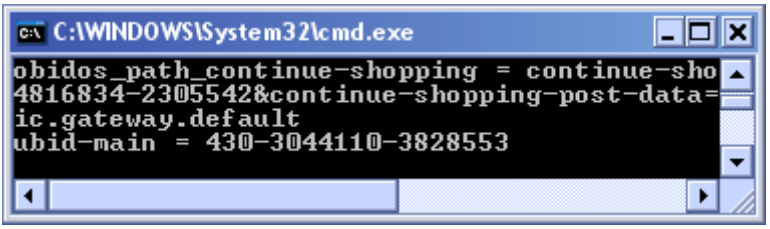
Вероятно всеки се е сблъсквал с т.нар. **cookies (бисквитки)** – малки текстови файлове на локалната машина на клиента, в които уеб-сървърите запазват различна информация за клиента, например потребителско име, парола, сесия и т.н. .NET Framework ни предлага лесен начин за преглеждане на cookies, записани от сървъра при изпълнение на заявката.

Преди да извикаме метода **GetResponse()**, трябва да създадем един празен обект на класа **CookieContainer** и да го присвоим на свойството **CookieContainer** на нашия **HttpWebRequest** обект. След изпълнение на заявката всички записани cookies се съхраняват в свойството **Cookies** на класа **HttpWebResponse** във вид на **CookieCollection**, която можем да обходим:

```
string url = "http://www.amazon.com/";
HttpWebRequest request =
   (HttpWebRequest) WebRequest.Create(url);
request.CookieContainer = new CookieContainer();
HttpWebResponse response =
   (HttpWebResponse) request.GetResponse();
foreach(Cookie ck in response.Cookies)
```

```
{
   Console.WriteLine("{0} = {1}", ck.Name, ck.Value);
}
```

Изпълнението на този код ни показва в текстов вид каква информация се запазва като cookies на локалната ни машина:



Други видове WebRequest и WebResponse

Класовете **HttpWebRequest**, **HttpWebResponse** и **FileWebRequest**, **FileWebResponse** се грижат за поддръжката на протоколите **http://**, **https://** и **file://**. Ако желаем да извършваме заявки по друг протокол на приложно ниво (например **ftp://**), можем да си напишем наследници на абстрактните класове **WebRequest** и **WebResponse** и да работим с тях. За целта трябва да помним няколко важни правила (например да не създаваме конструктор, а да разчитаме на метода **WebRequest.Create(…)**) и да не забравяме да свържем новия клас с желания вид протокол. Това става чрез метода **RegisterPrefix(string prefix, IWebRequestCreate create)** на класа **WebRequest**. Като първи аргумент подаваме именно префикса на протокола, с който ще свържем нашия нов тип, а като втори аргумент – на практика нашия клас, който да реализира метода **Create(…)** (чрез този метод ще се създават обектите на заявките) и да имплементира интерфейса **IWebRequestCreate**, който съдържа прототипа на този метод. Ако вече имаме регистриран за този префикс клас, ще получим изключение.

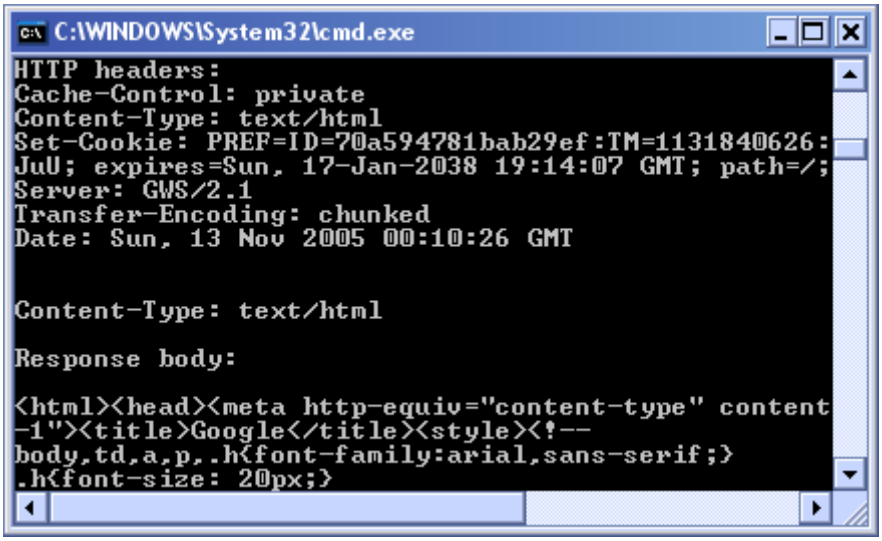
Работа с HTTP заявки – пример

Следната демонстрация показва накратко използването на **HttpWebRequest** и **HttpWebResponse**. Създаваме един **HttpWebRequest** обект, после изпълняваме **GET** заявка с него и извеждаме на екрана различната информация, съдържаща се в получения по този начин обект от класа **HttpWebResponse**. За тази цел:

1. Отваряме VS .NET и създаваме ново конзолно приложение.
2. Въвеждаме кода на програмата. Ще използваме заявка към сайта на Google, но това, разбира се, може да се промени лесно. Създаваме клас **HttpWebRequestWebResponse** и записваме в него основния метод на програмата:

```
static void Main(string[] args)
{
   string url = "http://www.google.com/";
   HttpWebRequest request =
     (HttpWebRequest) WebRequest.Create(url);
   HttpWebResponse response =
     (HttpWebResponse) request.GetResponse();

   WebHeaderCollection headers = response.Headers;
   Console.WriteLine("HTTP headers:\n{0}", headers);

   string contentType = response.ContentType;
   Console.WriteLine("Content-Type: {0}\n", contentType);

   Console.WriteLine("Response body:\n");
   Stream stream = response.GetResponseStream();
   using (stream)
   {
      StreamReader sr = new StreamReader(stream);
      string responseBody = sr.ReadToEnd();
      Console.WriteLine(responseBody);
   }
}
```

3. Стартираме приложението, за да изпълним заявката. Резултатът изглежда по следния начин:



Работа с електронна поща

В света дневно се обменят милиарди електронни писма. Несъмнено размяната на електронни писма е изключително популярен метод за комуникация. Нека разгледаме протоколите, свързани с изпращането и

получаването на електронна поща и как можем да изпращаме електронни съобщения със стандартните средства на .NET Framework.

Протоколи за изтегляне на електронната поща

Когато си създадете нова пощенска кутия, на специален компютър за вас се създава място, където могат да пристигат вашите писма. Основно има два метода да проверите съдържанието на това хранилище на електронни писма. Първият начин е чрез уеб интерфейс, при който писмата са представени пред вас под формата на уеб страница. Вторият начин е чрез използването на e-mail клиент като Microsoft Outlook, Pegasus или Eudora. За да могат тези клиенти да осъществят връзка със сървъра, на който се пазят писмата, се използва протоколът Post Office Protocol версия 3 (POP3) или Internet Message Access Protocol (IMAP).

Протоколът POP3

Протоколът POP3 основно позволява да изтегляте и изтривате писмата от пощенския сървър. Концепцията за работа с него предполага, че клиентът съхранява пощата си локално и от време на време се свързва със сървъра, изтегля новите писма и ги изтрива от там. Локално писмата могат да се подреждат по папки, примерно: входящи (Inbox), изходящи (Sent Items) и т. н.

Протоколът IMAP

Подходът с POP3 се оказва неподходящ, когато потребителят използва няколко различни машини за всекидневната си работа и трябва да чете пощата си от всяка от тях. Настъпва необходимостта пощата, организирана в папки, заедно с адресната книга да се съхраняват в централно хранилище, до което да се позволява отдалечен достъп.

За тази цел е създаден и протоколът IMAP. Той поддържа папки със съобщения и операции като добавяне на съобщение, преместване на съобщение, изтриване на съобщение. Има и други възможности като търсене по ключови думи в папките на сървъра.

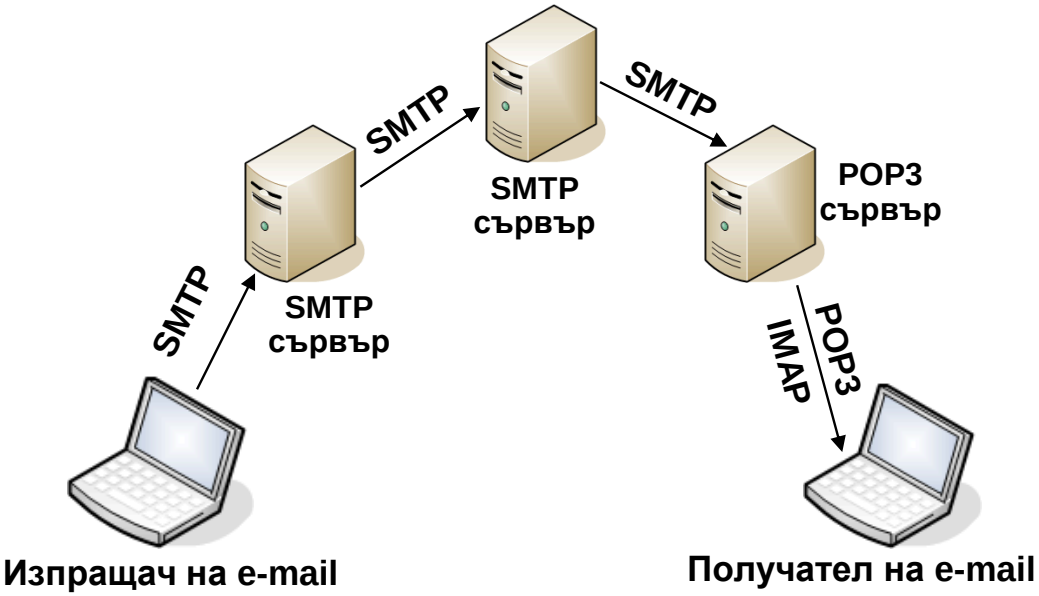
Изтегляне на електронната поща с .NET Framework

В .NET не е имплементирано изтеглянето на пощата нито с протокола POP3, нито с IMAP. Ако желаем да създадем приложение, което изтегля поща, трябва сами да си имплементираме работата по някой от тези протоколи.

Изпращане на електронна поща

За доставката на електронното писмо до получателя се използва мрежа от компютри, които могат да препращат съобщенията, докато те достигнат до крайната си цел. Тези компютри си комуникират посредством протокола SMTP – Simple Mail Transfer Protocol и се наричат SMTP сървъри. За да

осъществим връзка с някой SMTP сървър, е нужно да използваме протокола SMTP. Изпращайки писмо до който и да е SMTP сървър, сме сигурни, че това писмо ще достигне до получателя, защото SMTP сървъра има за цел да установи точно къде е получателят и да го изпрати избирайки оптимален път. Но кой SMTP сървър да използваме? Най-добрият вариант е да използваме SMTP сървъра на доставчика, предоставящ ни достъп до Интернет.



Изпращане на електрона поща с .NET Framework

За изпращане на електронно съобщение ще използваме класа **`System.Web.SmtpMail`**.

Най-лесният начин да изпратим електронно писмо

Ако искаме да изпратим простичко електронно писмо с помощта на .NET, можем да го направим с минимум усилия. Нужно е само да зададем SMTP сървър, към който да изпратим писмото. Това ще направим, използвайки статичното поле **`SmtpServer`** на класа **`SmtpMail`**. Същинското изпращане на съобщението става чрез статичния метод **`Send(…)`** на същия клас. Методът приема четири параметъра, съответно: адрес на подателя, адрес на получателя, полето "относно" (subject) и текста на съобщението. Възниква въпросът дали когато въвеждаме адреса на подателя, е възможно да въведем всеки електронен адрес, който желаем? Ако можем, нищо няма да ни пречи да изпращаме електронни писма от името на съседа, шефа или дори президента. В крайна сметка се оказва, че SMTP протоколът няма начин да провери дали сме въвели своя електронен адрес или не. Можем да въведем почти произволен адрес на подател. Някои SMTP сървъри не позволяват изпращането на електронни писма от несъществуващ домейн и това би предизвикало изключение при

изпълнение на метода за изпращане. Други SMTP сървъри биха променили адреса, така че изпращането да стане от съществуващ домейн.

Казаното дотук можем да илюстрираме със следния пример:

```
string from = "ivan@somemail.bg";
string to = "draganka@somemail.net";
string subject = "Cool Subject";
string body = "This is the message body";
SmtpMail.SmtpServer = "smtp.MyISP.com";
SmtpMail.Send(from, to, subject, body);
```

Особеното в случая е, че ако изпращането на съобщението не е успешно, ще се предизвика **System.Web.HttpException** изключение.

Формат на електронните съобщения

Всяко електронно писмо е в текстов формат и съдържа заглавна част и тяло. В заглавната част се съдържа служебна информация, а в тялото се съдържа текстът на съобщението. Заглавната част се състои от множество полета, предоставящи информация за писмото (наричани също хедъри или headers). Тези полета са текстови записи във вид **ключ: стойност**. Ето няколко примерни заглавни полета:

```
Return-Path: <doctora@gmail.com>
From: <doctora@gmail.com>
To: <kolio@kolev.net>
Subject: proba
Date: Sat, 29 Oct 2005 15:37:51 +0300
```

Някои от заглавните полета се добавят при съставяне на писмото. Такива са заглавните полета, обозначаващи получателя, изпращача, софтуера използван за съставяне на писмото и др. А има някои полета, които се добавят допълнително. Пътя на едно писмо може да мине през няколко различни SMTP сървъра и всеки сървър добавя по едно заглавно поле в писмото, обозначаващо, че писмото е минало през този сървър. Заради това може да видите в заглавната част на някое писмо няколко заглавни блока – това са полетата с ключ **Received**.

```
Received: from cyclone.host.bg (cyclone.host.bg
[217.160.253.243])
  by beeblebrox.host.bg (8.13.1/8.12.11) with ESMTP id
j9TCbulS025094
  for <doktora@kolev.com>; Sat, 29 Oct 2005 15:37:57 +0300
Received: from nproxy.gmail.com (nproxy.gmail.com
[64.233.182.202])
  by cyclone.host.bg (Postfix) with ESMTP id C5B741939B4
  for <doktora@kolev.com>; Sat, 29 Oct 2005 15:40:43 +0300
(EEST)
Received: by nproxy.gmail.com with SMTP id x4so204908nfb
```

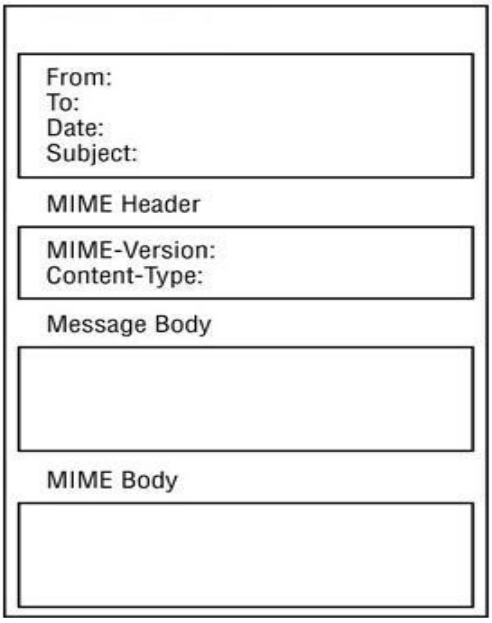
```
        for <doktora@kolev.com>; Sat, 29 Oct 2005 05:37:54 -0700
(PDT)
```

Както видяхме по-горе, може да изпратим писмо с изпращач всеки валиден електронен адрес. Благодарение на тези заглавни части, показващи пътя на писмото, в повечето случаи можем да преценим дали дадено писмо е изпратено от адреса, който е написан като изпращач, или от някой злонамерен хакер.

Multipurpose Internet Mail Extension (MIME)

Както казахме по-горе, едно електронно писмо е изградено само от текст. Затова е необходимо да можем да преобразуваме прикачените файлове до текст и съответно получателят да може да преобразува текста до двоични файлове, без да има загуби. За тези цели е създадено разширението Multipurpose Internet Mail Extension (MIME).

MIME е стандартен формат за съобщения. Използва се в различни системи за обмяна на съобщения и в частност при електронната поща.



MIME добавя 5 нови заглавни полета в края на заглавната част на електронното писмо и всеки прикачен файл (преобразуван до текст) се добавя в края на писмото.

Един MIME документ се състои от съвкупност от файлове (документи), всеки, от които е кодиран като текст (например чрез кодиране BASE64) и има зададен тип (Content-Type) и съдържание. Типовете в MIME стандарта могат да бъдат най-различни: PDF документи, ZIP архиви, HTML страници, картинки, музика и т.н. За всеки тип си има уникален идентификатор.

MIME – пример

Ще даден един пример за електронно писмо, което е изградено чрез MIME стандарта. То съдържа 3 документа: писмото в текстов формат, писмото в HTML формат и файл, който е прикрепен към писмото (attachment):

```
From: "Svetlin Nakov" <nakov@nospam.somewhere.org>
To: "Mincho Penchev <mincho@mail.penchev.org>"
Subject: Hello, Mincho
Date: Sun, 16 Jul 2005 11:45:25 +0300
MIME-Version: 1.0
Content-Type: multipart/mixed;
  boundary="----=_NextPart_000_0003_01C6A8CD.53FD2960"
X-Priority: 3
X-MSMail-Priority: Normal
X-MimeOLE: Produced By Microsoft MimeOLE V6.00.3790.2663

This is a multi-part message in MIME format.

------=_NextPart_000_0003_01C6A8CD.53FD2960
Content-Type: multipart/alternative;
  boundary="----=_NextPart_001_0004_01C6A8CD.53FD2960"

------=_NextPart_001_0004_01C6A8CD.53FD2960
Content-Type: text/plain;
  charset="windows-1251"
Content-Transfer-Encoding: 8bit

Здравей, Минчо!

Пише ти Светлин Наков. Поздрави от София. Искам да те поканя на
един семинар за ИТ специалисти. Виж приложената покана.

Svetlin Nakov
National Academy for Software Development
academy.devbg.org

------=_NextPart_001_0004_01C6A8CD.53FD2960
Content-Type: text/html;
  charset="windows-1251"
Content-Transfer-Encoding: 8bit

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html><head>
<meta http-equiv="Content-Type" content="text/html;
charset="windows-1251"></head>
<body>
<p><font face="Arial">Здравей, Минчо!<br><br>
Пише ти Светлин Наков. Поздрави от София. Искам да те поканя на
един семинар за ИТ специалисти. Виж приложената покана.<br><br>
<font color="#000080"><b>Svetlin Nakov</b></font><br>
```

```
<font size="2"><font color="#000080">National Academy for
Software Development</font><br>
<a href="http://academy.devbg.org/">
academy.devbg.org</a></font></font></p>
</body></html>

------=_NextPart_001_0004_01C6A8CD.53FD2960--

------=_NextPart_000_0003_01C6A8CD.53FD2960
Content-Type: application/msword;
   name="BARS-seminar-26-July-2005-pokana.doc"
Content-Transfer-Encoding: base64
Content-Disposition: attachment;
   filename="BARS-seminar-26-July-2005-pokana.doc"

0M8R4KGxGuEAAAAAAAAAAAAAAAAAAAAAPgADAP7/CQAGAAAAAAAAAAAAAAACAAAAf
gAAAAAAAAA
... ... ...
AAsAAAAAAAAAHhAAAAEAAAA4AAAARGF0YWJhc2UgUHJvZ3JhbW1pbmcgQmVzdCBQc
mFjdGljZXMg
LSBUZWNobmljYWwgU2VtaW5hcgAMEAAAAgAAAB4AAAAGAAAA

------=_NextPart_000_0003_01C6A8CD.53FD2960—
```

Съставяне на електронно писмо

Вече видяхме как можем лесно да изпратим електронно писмо, но този начин за работа не ни предоставя много възможности. Сега ще се запознаем с класа **System.Web.Mail.MailMessage**. Този клас ни позволява да създадем съобщение, използвайки повече възможности от показаното по-горе.

Можем да използваме полетата **From**, **To**, **Subject** и **Body**, за създаване на писмо, аналогично на писмото, изпратено по лесния начин.

Интересно е, че с помощта на този клас можем да променим приоритета на писмото или да укажем схемата на кодиране. Също така можем да укажем дали писмото е в HTML или в текстов формат. Ако изпращаме писма в HTML формат, повечето клиенти ще могат да ги визуализират, но трябва да знаем, че има и клиенти, които нямат такава възможност.

```
MailMessage message = new MailMessage();
message.From = "doktor_ivanov@somemail.com";
message.To = "kaka_tonka@mail.bg";
message.Subject = "Zdrasti!";
message.BodyFormat = MailFormat.Html;
message.Body = @"<html><body><h1>Sreshtata dovechera se " +
   "otmenia.</h1></body></html>";
SmtpMail.SmtpServer = "mail.interbgc.com";
SmtpMail.Send(message);
```

Прикачени файлове

За да изпратим прикачен файл в .NET Framework, можем да използваме класа **System.Web.Mail.MailAttachment**. В конструктора на класа указваме желания файл и го свързваме с писмото по следния начин:

```
MailMessage message = new MailMessage();
message.From = "didi@kaval.com";
message.To = "bobby@duduk.net";
message.Subject = "Hi, Bobby!";
message.Body = "Here's my picture!!";
MailAttachment attachment = new
  MailAttachment(@"c:\images\logo.gif");
message.Attachments.Add(attachment);
SmtpMail.Send(message);
```

Автентикация пред SMTP сървър

С нарастването на нежеланата поща (spam), се оказа, че много SMTP сървъри се използват за изпращане на нежелана поща. Наложи се SMTP сървърите да предлагат услугите си само на доказали самоличността си потребители. Това обикновено става чрез потребителско име и парола. В .NET Framework 1.0 няма поддръжка на такъв вид автентикация. Във версия 1.1, макар и по малко странен начин, вече е въведено автентикирането пред SMTP сървъра. Използва се класът **MailMessage**:

```
MailMessage message = new MailMessage();
message.Fields.Add("http://schemas.microsoft.com/cdo/configuratio
  n/"+ "smtpauthenticate", 1);
message.Fields.Add("http://schemas.microsoft.com/cdo/configuratio
  n"+  "/sendusername", "doktora");
message.Fields.Add("http://schemas.microsoft.com/cdo/configuratio
  n/"+ "sendpassword", "1234567");
```

В илюстрирания пример добавяме три записа. Първият запис указва начина, използван за автентикация. Вторият запис указва потребителското име (в някои случаи то съвпада с електронния адрес) – в случая потребителското име е "doktora". Третият запис задава паролата, в случая това е "1234567".

Грешката "Could not access CDO Object"

Както споменахме по-горе, ако изпращането на пощата е невъзможно поради някаква причина, ще бъде предизвикано изключение от тип **System.Web.HttpException**. Много заблуждаващо е, когато разглеждаме изключението и в полето **Message** е изписано **"Could not access CDO Object"**. По-начинаещ програмист би могъл да се заблуди, че това е грешката, което не е вярно. Истинското съобщение можем да извлечем по следния начин:

```
try
{
   SmtpMail.Send(message);
}
catch (System.Web.HttpException ex)
{
   Console.WriteLine("Unable to send message: {0}",
     ex.InnerException.InnerException.Message);
}
```

Виждате, че проблемите с класа **MailMessage** са много, но докато Microsoft не добавят истински клас за пращане на e-mail, ни остават възможностите да напишем сами SMTP клас, да си намерим такъв от трети доставчици или да се примирим с неудобствата на класа **MailMessage**.

Упражнения

1. Опишете 7-те слоя от OSI мрежовия модел.
2. Обяснете понятията: IP адрес, DNS, порт, мрежов интерфейс, TCP, UDP и сокет връзка. Каква е разликата между протоколите TCP и UDP?
3. Опишете основните мрежови услуги в Интернет, какви протоколи използват и кои TCP портове.
4. Реализирайте Windows Forms приложение, наподобяващо по функционалност инструмента telnet. Приложението трябва да поддържа свързване към отдалечен TCP сървър, изпращане и приемане на данни и прекъсване на установена сокет връзка. Използвайте нишки, за да направите възможно едновременното изпращане и получаване на данни.
5. Реализирайте многопотребителски сървър за разговори (chat server). Сървърът трябва да работи по протокол TCP и да позволява регистриране на потребители и изпращане на съобщения между потребителите. Упътване: Реализирайте 2 команди: **USER <username>** и **SEND <username> <message>**. Работете с текстови потоци. Направете всяка команда да е точно един текстов ред и сървърът да връща при всяка команда 1 текстов ред отговор (**OK** или **ERR**). За всеки потребител използвайте 2 TCP връзки (и 2 нишки, които да ги обслужват) – едната за приемане на команди, а другата за изпращане на съобщения. Съобщенията, които не са изпратени, съхранявайте в блокираща обща опашка (shared queue).
6. Реализирайте Windows Forms клиент за сървъра за разговори (chat server) от предната задача.
7. Да се реализират UDP версии на chat сървъра и клиента за него: вместо по TCP връзки всички команди и съобщения трябва да се пращат чрез UDP пакети.

8. Напишете Windows Forms приложение, което изпълнява DNS заявки (преобразува от име на машина към IP адрес и обратното).
9. Да се реализира приложение, което извлича главната страница от сайта http://www.devbg.org/ и отпечатва всички препратки (hyperlinks). За извличане на препратките използвайте подходящ регулярен израз.
10. Да се напише паяк (Web spider) за събиране на e-mail адреси. Паякът работи така: Започва от даден URI адрес в Интернет и извлича съдържанието му. Чрез подходящи регулярни изрази извлича от него всички e-mail адреси и всички препратки (hyperlinks). Препратките канонизира (прави ги на URI адреси) и поставя в опашка. Докато не бъде спрян, паякът обработва по същия начин поредния URL адрес от опашката. Размерът на опашката да се ограничи до 50 000. За по-голямо бързодействие да се работи с 10 нишки едновременно. За извличане на даден URI адрес използвайте класа WebClient. За комбиниране на абсолютен и релативен URI използвайте конструктора на Uri класа.
11. Да се състави програма, която изпраща дадено e-mail съобщение (записано в текстов файл) на списък от получатели (зададени чрез текстов файл). SMTP сървърът и подателят на e-mail съобщението трябва да се задават от конфигурационния файл на приложението.
12. Да се реализира приложение, което стои в "task bar" областта и изпраща на всеки 10 минути предварително зададен файл на предварително зададен e-mail адрес.

Използвана литература

1. Ивайло Христов, Мрежово и Интернет програмиране – http://www.nakov.com/dotnet/lectures/Lecture-17-Internet-Access-v1.0.ppt
2. Светлин Наков, Интернет програмиране с Java, Faber, 2004, ISBN 954-775-305-3 – http://www.nakov.com/books/inetjava/
3. Richard Blum, C# Network Programming, Sybex, 2003, ISBN 0782141765
4. MSDN Training, Programming with the Microsoft® .NET Framework (MOC 2349B), Module 11: Internet Access
5. OSI model – Wikipedia, the free encyclopedia – http://en.wikipedia.org/wiki/OSI_model

НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

Лекторите

» ***Светлин Наков*** е автор на десетки технически публикации и няколко книги, свързани с разработката на софтуер, заради което е търсен лектор и консултант.

Той е разработчик с дългогодишен опит, работил по разнообразни проекти, реализирани с различни технологии (.NET, Java, Oracle, PKI и др.) и преподавател по съвременни софтуерни технологии в СУ "Св. Климент Охридски".

През 2004 г. е носител на наградата "**Джон Атанасов**" на президента на България Георги Първанов.

Светлин Наков ръководи обучението по Java технологии в Академията.

» ***Мартин Кулов*** е софтуерен инженер и консултант с дългогодишен опит в изграждането на решения с платформите на Microsoft.

Мартин е опитен инструктор и сертифициран от Майкрософт разработчик по програмите MCSD, MCSD.NET, MCPD и MVP и международен лектор в световната организация на .NET потребителските групи INETA.

Мартин Кулов ръководи обучението по .NET технологии в Академията.

Академията

» **Национална академия по разработка на софтуер** (**НАРС**) е център за професионално обучение на софтуерни специалисти.

» **НАРС** провежда **БЕЗПЛАТНО** курсове по разработка на софтуер и съвременни софтуерни технологии в София и други градове.

» Предлагани специалности:

- **Въведение в програмирането (с езиците C# и Java)**
- **Core .NET Developer**
- **Core Java Developer**

» **Качествено обучение** с много **практически проекти** и индивидуално внимание за всеки.

» **Гарантирана работа!** Трудов договор при постъпване в Академията.

» БЕЗПЛАТНО!

Учите безплатно във въведителните курсове и по стипендии от работодателите в следващите нива.

http://academy.devbg.org

Глава 19. Отражение на типовете (Reflection)

Автор

Димитър Канев

Необходими знания

- Базови познания за .NET Framework и Common Language Runtime (CLR)
- Базови познания за езика C#
- Базови познания за MSIL

Съдържание

- Какво е Global Assembly Cache?
- Какво е Reflection?
- Зареждане на асемблита
- Извличане информация за асембли
- Премахване на асемблита от паметта
- Изучаване членовете на тип
- Извличане на методи и параметрите им
- Извличане на параметрите на метод
- Динамично извикване на методи
- Reflection Emit

В тази тема ...

В настоящата тема ще представим понятието Global Assembly Cache (GAC) и отражение на типовете (reflection). Ще се запознаем с начините за зареждане на асембли. Ще покажем как можем да извлечем информация за типовете в дадено асембли и за членовете в даден тип. Ще разгледаме начини за динамично извикване на членове от даден тип. Ще разберем как можем да създадем едно асембли, да дефинираме типове в него и да го запишем във файл по време на изпълнение на дадена програма.

Какво е Global Assembly Cache?

Global Assembly Cache (GAC) е централно хранилище на споделени асемблита, до които се осъществява достъп от много приложения. Всяка .NET инсталация има един Global Assembly Cache. Обикновено той се намира в директорията:

```
C:\Windows\Assembly
```

Директорията на GAC има определена структура, съставена от множество поддиректории, в които се намират манифест файловете на съхраняваните асемблита. Името на дадена поддиректория в GAC се генерира спрямо името на съхраняваното асембли в нея. По този начин в GAC се подържа връзка между споделено асембли и поддиректория.

Освен споделени асемблита, в отделна секция на GAC, се съхранява прекомпилирани асемблита, при чието изпълнение, средата не извиква всеки път JIT компилатора, а изпълнява директно прекомпилирания код.

В Global Assembly Cache се пази и свален от Интернет или локални мрежи код на асемблита. Средата ограничава достъпа до тях, като ги съхранява в частна секция на GAC.

Съхраняваните споделени асемблита в GAC, задължително трябва да са силно именувани, което ги идентифицира уникално. Поставянето на асемблитата в GAC става чрез инсталиране.

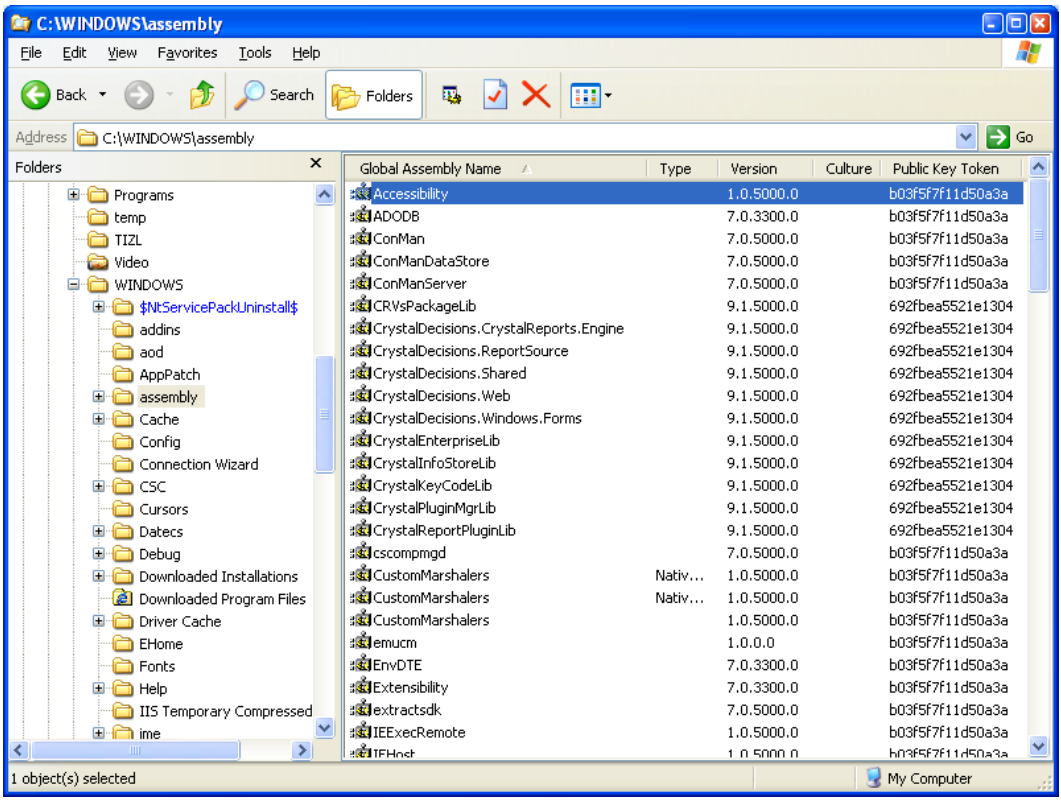
Инсталиране на асемблита в GAC

За да се използва частно асембли от дадено приложение е достатъчно то да се копира в директорията, в която се намира приложението. Споделените асемблита не могат да се копират направо в GAC - те трябва да се инсталират.

Инсталирането на асемблита в GAC става с помощта на инструменти за инсталация, които познават структурата му. Един от най-често използваните инструменти за инсталиране, деинсталиране и показване на асемблита в GAC е Global Cache Accessibly Utility (**`gacutil.exe`**).

Много потребители предпочитат да използват Windows Explorer за инсталиране на асемблита. Това става по следния начин:

1. Стартираме Windows Explorer.
2. Отваряме директорията **`C:\Windows\Assembly`**.
3. Взимаме файла, съдържащ манифеста на асемблито, което искаме да инсталираме и го пускаме в прозореца на Windows Explorer и асемблито ще бъде инсталирано.



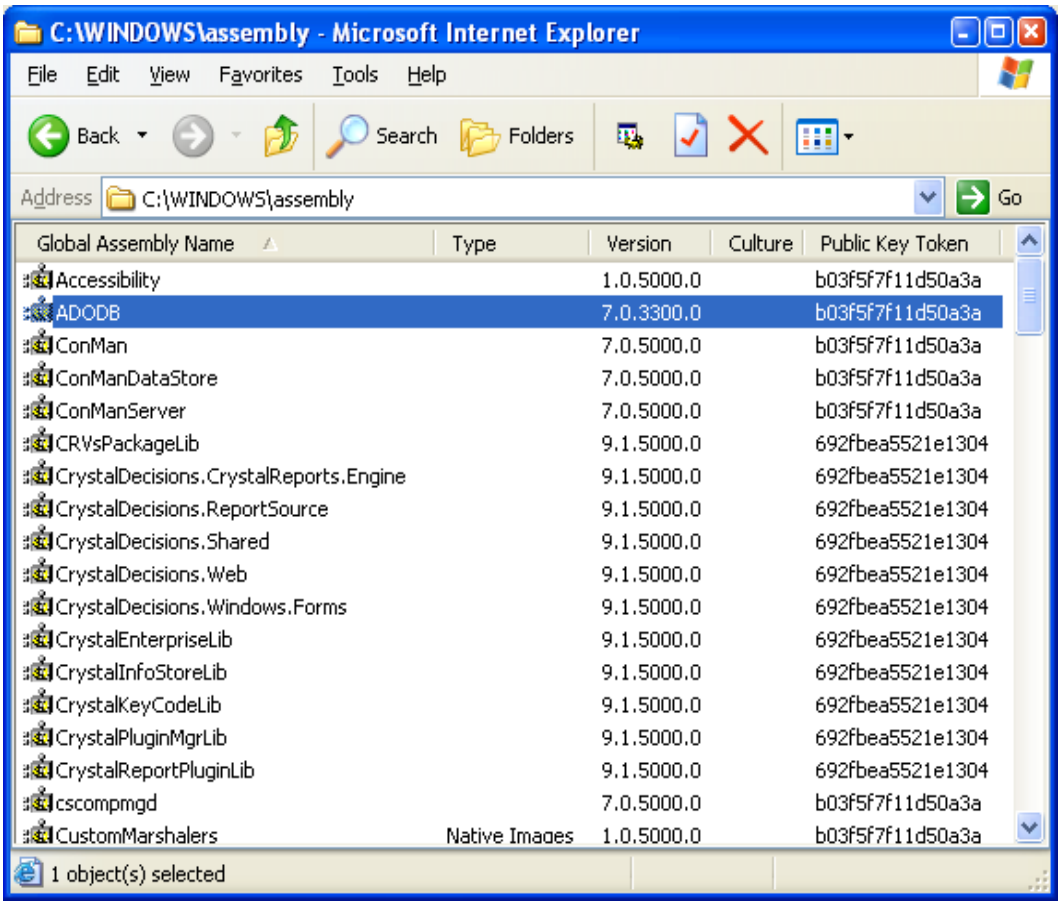
Поддръжка на много версии

Една от основните цели на GAC е поддържането на множество версии на едно асембли. Едновременното поддържане на няколко инсталирани версии на едно и също асембли се използва от CLR средата при управлението на зареждането на асемблита. Инсталирането на нова версия на едно асембли не влияе на вече инсталираните приложения. При стартиране на приложение, използващо споделено асембли, CLR средата проверява с коя версия е изградено приложението и зарежда споделеното асембли с подходящата версия.

Преглед на GAC през Windows Explorer

С помощта на Windows Explorer може да се разгледа съдържанието на GAC в удобен и разбираем вид. В настоящия пример ще бъде демонстрирано как става това.

1. Стартираме Windows Explorer.
2. Отваряме директорията **`C:\Windows\Assembly`** и виждаме инсталираните асемблита в GAC на нашия компютър:



3. Разглеждаме асемблитата от списъка.

Всяко асембли инсталирано в GAC се показва в Windows Explorer със своите характеристики:

- Global Assembly Name - името на асемблито
- Type - типа на асемблито
- Version – версия на асемблито
- Culture - култура на асемблито
- Public Key Token - публичен ключ на асемблито

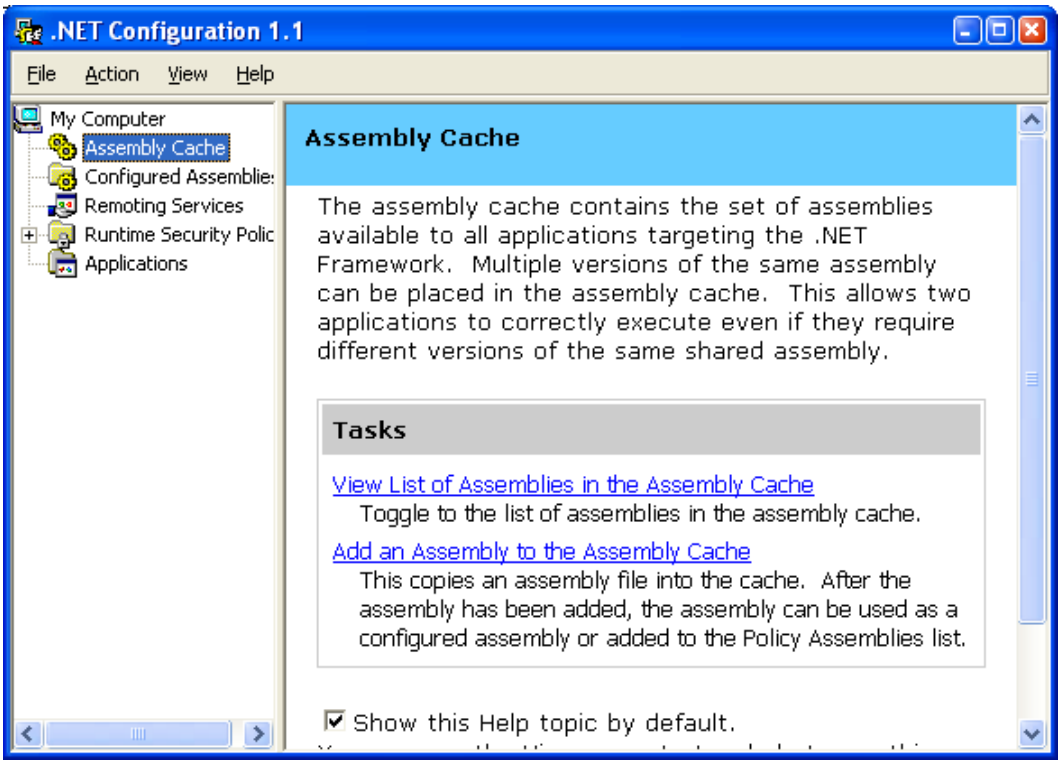
Допълнителни характеристики за дадено асембли са достъпни при избор на Properties от контекстното меню, което се показва при натискане на десен бутон на мишката върху запис за асембли. Допълнителните характеристики се показват в диалогов прозорец:

Дадено асембли може да се изтрие от GAC като се избере Delete от контекстното меню, което се показва при натискане на десен бутон на мишката върху запис за асембли.

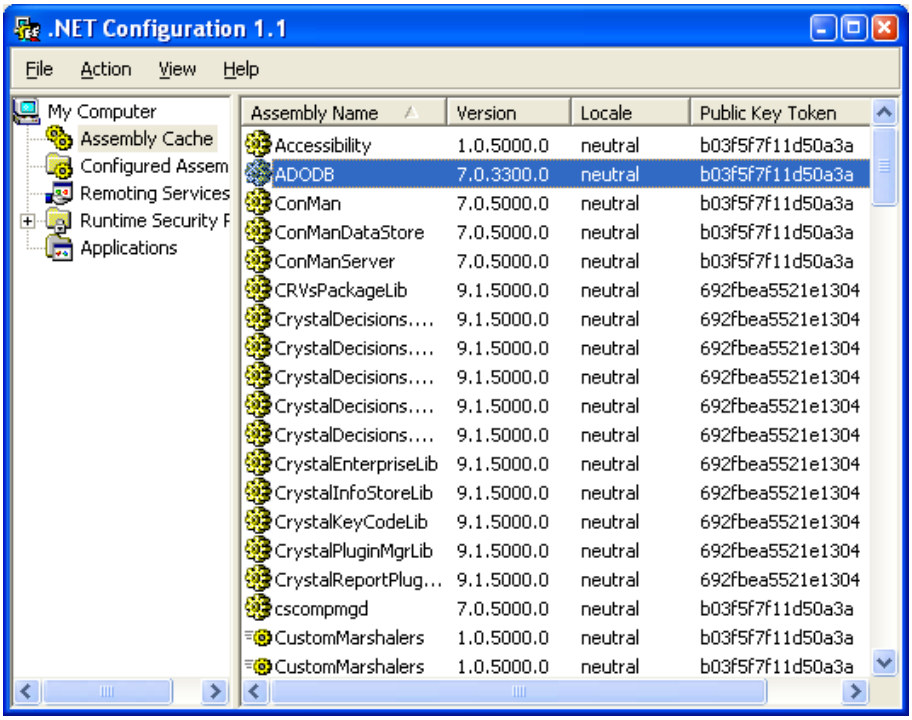
Преглед на GAC през Administrative Tools

Съдържанието на GAC може да се разгледа и с помощта на Microsoft .NET Framework Configuration.

1. Отваряме Control Panel --> Administrative Tools --> Microsoft .NET Framework Configuration
2. От полето My Computer избираме "Assembly Cache".



3. Последваме хипервръзката View List of Assemblies in the Assembly Cache и разглеждаме асемблитата от списъка.



Отражение на типовете

Отражението на типовете (reflection) е един нов механизъм, предоставен от платформата .NET Framework, даващ възможност за получаване на информация за типовете по време на изпълнение на програмата. Това позволява проектирането на динамично разширяващи се приложения, към които лесно могат да се добавят типове на други програмисти и компании.

В някаква форма отражение (reflection) има във всички управлявани платформи като Java, Perl и др.

Какво е Reflection?

Reflection е механизъм, даващ следните възможност на .NET приложенията:

- да изучават метаданните на асемблита
- да изучават типовете в дадено асембли
- динамично да извикват методи
- динамично да създават нови асемблита, да ги изпълняват и да ги запазват като файл

С помощта на отражение, дадено приложение може да зареди динамично дадено асембли (DLL файл), да извлече от него даден тип, да го инстанцира динамично и да му извика методите. Механизмът на отражение позволява да се добавят по време на изпълнение към даден метод MSIL инструкции, след което той да се изпълни. Възможностите предоставени от този механизъм широко се използват при писането на компилатори и интерпретатори на скрипт езици. С помощта на reflection могат да се пишат плъгини (plugins).

Зареждане на асемблита

Отражението на типовете (reflection) може да се използва, когато по време на изпълнение, дадено приложение трябва да получи информация за асембли или тип. Това може да се постигне чрез създаване на **`System.Reflection.Assembly`** обект, идентифициращ заредено асембли и извикването на предоставените от него методи и свойства. Класът **`System.Reflection.Assembly`** представя асембли в CLR средата.

`System.Reflection.Assembly` обект може да се създаде чрез извикването на методите **`Assembly.Load(…)`** или **`Assembly.LoadFrom(…)`**.

Зареждане чрез Assembly.Load(...)

Методът **`System.Reflection.Assembly.Load(…)`** приема като параметър име на асембли или обект от тип **`System.Reflection.AssemblyName`**, който описва асемблито. При извикване на този метод, асемблито първо се търси в GAC, след това в базовата директория на приложението и накрая

в частните пътища. Ако не бъде намерено търсеното асембли, се подава изключение **FileNotFoundException.** Методът връща зареденото асембли.

Зареждане чрез Assembly.LoadFrom(...)

Методът **System.Reflection.Assembly.LoadFrom(…)** приема като параметър пътя на файла на асемблито, което искаме да заредим. При извикването на този метод, CLR средата зарежда указания от параметъра файл. Вътрешно се извиква **Assembly.Load(…)**. Ако не се намери търсеният файл, се подава изключението **FileNotFoundException**. Методът връща зареденото асембли. Този метод е по-бавен от **Assembly.Load(…)**.

Извличане информация за асембли

Извличането на информация за асембли става с помощта на свойствата на класа **System.Reflection.Assembly**:

- **FullName** – съдържа пълното име на асемблито, включващо версия, култура и ключ (Public Key Token).
- **Location** – съдържа пътят, от където е заредено асемблито.
- **EntryPoint** – съдържа метода, от който ще започне изпълнението на асемблито.
- **GlobalAssemblyCache** – булева стойност, която показва дали асемблито е било заредено от GAC.

Двукратно зареждане на асембли – пример

В настоящия пример се демонстрира двукратно зареждане на асембли и се илюстрира ходът на изпълнение на следния програмен код:

```
using System;
using System.Reflection;
using System.IO;

class AssemblyDoubleLoad
{
   static private void ShowAllAsemblies()
   {
      foreach(Assembly assembly in
         AppDomain.CurrentDomain.GetAssemblies())
      {
         Console.WriteLine(assembly.FullName);
         Console.WriteLine(assembly.Location);
         Console.WriteLine();
      }
   }

   static void Main()
   {
```

```
      // List all assemblies at the beginning
      Console.WriteLine(" ------------------- In the beginning --
        ------------------");
      ShowAllAsemblies();

      // Load mscorlib.dll with Assembly.Load() and list all
      // asemblies
      Assembly.Load("mscorlib.dll");
      Console.WriteLine("------ After loading Mscorlib
        with Assembly.Load() ------");
      ShowAllAsemblies();

      // Load mscorlib.dll from a different path
      Assembly.LoadFrom(@"..\..\mscorlib.dll");
      Console.WriteLine(" --- After loading mscorlib.dll from a
        different path ---");
      ShowAllAsemblies();
   }
}
```

След изпълнение на примера се получава следният резултат:

```
C:\dot_net_book\Lecture-18-Reflection\Demo-2-DoubleLoad\bin\Debug\Demo-2-DoubleLoad....
 ------------------- In the beginning --------------------
mscorlib, Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
c:\windows\microsoft.net\framework\v1.1.4322\mscorlib.dll

Demo-2-DoubleLoad, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
C:\dot_net_book\Lecture-18-Reflection\Demo-2-DoubleLoad\bin\Debug\Demo-2-DoubleL
oad.exe

------ After loading Mscorlib with Assembly.Load() ------
mscorlib, Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
c:\windows\microsoft.net\framework\v1.1.4322\mscorlib.dll

Demo-2-DoubleLoad, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
C:\dot_net_book\Lecture-18-Reflection\Demo-2-DoubleLoad\bin\Debug\Demo-2-DoubleL
oad.exe

 --- After loading mscorlib.dll from a different path ---
mscorlib, Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
c:\windows\microsoft.net\framework\v1.1.4322\mscorlib.dll

Demo-2-DoubleLoad, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
C:\dot_net_book\Lecture-18-Reflection\Demo-2-DoubleLoad\bin\Debug\Demo-2-DoubleL
oad.exe

mscorlib, Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
c:\dot_net_book\lecture-18-reflection\demo-2-doubleload\mscorlib.dll
```

Описание на примера

Класът **AssemblyDoubleLoad** от примера има един статичен метод **ShowAllAsemblies(…)**, който извежда в конзолата пълните имена и местоположения на всички заредени асемблита в текущия **AppDomain** на приложението.

Най-напред в примера се извежда информация за асемблитата, заредени от CLR средата при стартиране на примера. Това са асемблитата

mscorlib.dll и **Demo-2-DoubleLoad.exe**. Асемблито **mscorlib.dll** съдържа повечето системни типове от пространството **System** на .NET Framework.

След това изрично се прави опит за зареждане на асемблито **mscorlib.dll** с помощта на метода **Assembly.Load(…)**. Този метод приема един параметър, името на асемблито, което искаме да заредим. След извикването на **Assembly.Load(…)** пак се извежда информация за заредените асемблита. В този случай не се зарежда ново асембли в паметта, защото CLR първо проверява дали поисканото асембли не е вече заредено и ако е така не го зарежда втори път.

По подразбиране в един домейн на приложението (AppDomain) едно асембли се зарежда само веднъж.

За да заредим два пъти асемблито **mscorlib.dll** го копираме в друга директория, за да заблудим CLR, че асемблитата са различни. В случая това е директорията на проекта.

Извикваме отново метода **Assembly.LoadFrom(…)** с параметър пътя до текущата директория на примера и името на асемблито (**mscorlib.dll**), което води до повторното му зареждане. При отпечатване на информацията за заредените асемблита в приложението се вижда, че **mscorlib.dll** е заредено един път при стартиране на приложението и втори път от текущата директория на приложението.

Премахване на асемблита от паметта

В CLR средата не се поддържа премахване на конкретно асембли от паметта. От паметта може да се премахнат всички асемблита, заредени в даден **AppDomain**. Това става с помощта на статичния метод **AppDomain.Unload(…)**, който приема референция към **AppDomain**, чиито асемблита искаме да премахнем от паметта. Използването на този метод не се препоръчва поради голямата вероятност от грешки.

Изучаване на типовете в асембли

При някои приложения е важно да се знае дали определен тип съществува в дадено асембли и какви методи и свойства предлага той. Тази информация може да не е достъпна по време на компилиране на асемблито, но да се знае по време на неговото изпълнение. Тогава се налага използването на отражение за динамично изучаване на типовете в дадено асембли. Динамичното извличане на информация за типовете не е от най-бързите операции и затова е препоръчително да се използва само, когато е абсолютно необходимо. При този подход за извличане на информация за типове, може да се допуснат грешки, свързани с безопасността на типовете, които компилаторът не може да открие и поправи.

Класът System.Type

Класът **System.Type** е абстрактен базов клас, наследник на класа **System.Reflection.MemberInfo**. Този клас представя даден тип от Common Type System (CTS) и предоставя възможност за поучаване на всичките му членове:

- полета
- методи
- свойства
- събития
- вложени типове

Класът **System.Type** е основен за механизма на отражение и предоставя множество свойства за достъп до метаданните на даден тип. Ще разгледаме някои от свойствата, дефинирани в класа **System.Type**:

- **BaseType –** връща родителския тип на текущия тип
- **Attributes –** връща атрибутите, свързани с текущия тип
- **FullName –** връща пълното име на текущия тип
- **IsAbstract –** връща **true**, ако типът е абстрактен
- **IsArray –** връща **true**, ако типът е масив
- **IsByRef –** връща **true**, ако типът е референтен
- **IsClass –** връща **true**, ако типът е клас
- **IsCOMObject –** връща **true**, ако типът е COM обект
- **IsEnum –** връща **true**, ако типът представлява изброен тип (енумерация)
- **IsInterface –** връща **true**, ако типът е интерфейс
- **IsPublic –** връща **true**, ако типът е деклариран като публичен

Получаване на System.Type обект

Има различни начини за получаване на **System.Type** обект. Един от тях е с помощта на метода **GetType()** на класа **System.Object**. При извикването на този метод CLR средата връща референция към типа на указания обект. По този начин за всеки обект може да се получи неговия тип. Следващият фрагмент показва как се извлича **Type** обект от променлива от тип **double:**

```
double d = 0.2;
Type t = d.GetType();
```

Класът **System.Type** предлага няколко предефинирани версии на статичния метод **GetType(…)**, приемащи като параметър **string**. Следващият фрагмент показва как се извлича **Type** обект с помощта на **System.Type.GetType(…)**:

```
Type t = Type.GetType("System.Double");
```

Този метод се използва от всички .NET езици и затова при използването му не може да му подадем като параметър **C#** псевдоним на тип (например **int**, **float**, **string**, ...). Трябва да се подава пълното име на типа, който искаме да получим.

Класът **System.Assembly** предлага метода **GetTypes()**, с чиято помощ могат да се получат всички типове от дадено асембли.

Изучаване членовете на тип

Извличането на членовете на даден тип става с помощта на следните методи, дефинирани в класа **System.Type**:

- **GetConstructors(…)** – връща конструкторите на текущия тип.
- **GetEvents(…)** – връща дефинираните или наследени събития на текущия тип.
- **GetFields(…)** – връща полетата на текущия тип (дефинирани в типа или наследени).
- **GetInterfaces(…)** – връща дефинираните и наследени интерфейси на текущия тип.
- **GetMembers(…)** – връща всички членове (полета, събития, свойства и др.) дефинирани в типа или наследени.
- **GetMethods(…)** – връща методите дефинирани в типа или наследени.
- **GetProperties(…)** – връща свойствата дефинирани в типа или наследени.
- **InvokeMember(…)** – извиква указан член на текущия тип. Може да се използва за извикване на конструктор или метод, за промяна на поле или свойство, както и за други по-сложни действия.
- **IsInstanceOfType(…)** – връща **true**, ако посоченият обект е инстанция на текущия тип.
- **System.Type.FindMembers(…)** – връща по зададен филтър членовете от определен вид на даден тип.

Всички разгледани методи на класа **System.Type** (с изключение на **IsInstanceOfType(…)**) връщат масив с референции от тип **System.Reflection.MemberInfo** или негов наследник. Класът **MemberInfo** е базов клас на всички класове, представляващи видовете членове на даден тип и

на класа **System.Type**. По-нататък в настоящата тема ще разгледаме по-подробно класовете за видовете членове на даден тип и самия клас **MemberInfo**. Важно е засега да се знае, че класът **MemberInfo** предлага следните свойства, които са общи за всички членове на даден тип:

- **MemberInfo.DeclaringType** – връща **System.Type** обект, който отразява типа, в който е дефиниран члена.
- **MemberInfo.MemberType** – връща вида на члена (поле, метод, тип, свойство, конструктор или събитие).
- **MemberInfo.Name** – връща името на члена или типа като **string**.
- **MemberInfo.ReflectedType** – връща **System.Type** обект, който е използван за получаването на този обект.

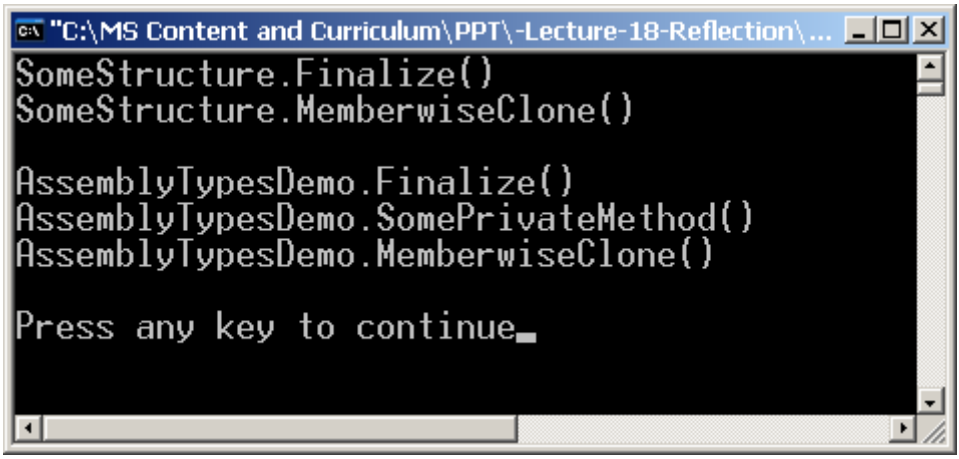
Изучаване членовете на тип – пример

В настоящия пример се демонстрира извличане на имената на типовете от асембли и имената на членовете на даден тип по зададен филтър и се илюстрира ходът на изпълнение на следния програмен код:

```
using System;
using System.Reflection;

struct SomeStructure
{
}

class AssemblyTypesDemo
{
   public void SomePublicMethod()
   {
      // Some code
   }

   private void SomePrivateMethod()
   {
      // Some code
   }

   private static void SomeStaticMethod()
   {
      // Some code
   }

   static void Main()
   {
      Assembly currentAssembly;
      currentAssembly = Assembly.GetExecutingAssembly();

      foreach (Type type in currentAssembly.GetTypes())
```

```
      {
         MemberInfo[] members = type.FindMembers(
            MemberTypes.Method,
            BindingFlags.NonPublic | BindingFlags.Instance,
            Type.FilterName,
            "*");
         foreach (MemberInfo member in members)
         {
            Console.WriteLine("{0}.{1}()", type.Name, member.Name);
         }

         Console.WriteLine();
      }
   }
}
```

След изпълнение на примера се получава следният резултат:



Описание на примера

В класа **AssemblyTypesDemo** за целите на примера са дефинирани освен методът **Main(…)** един публичен метод **SomePublicMethod(…)** и два частни метода - **SomePrivateMethod()** и **SomeStaticMethod()**, като вторият е статичен. Дефинирана е и външна за класа **AssemblyTypesDemo** структура - **SomeStructure**.

При стартиране на примера чрез **Assembly.GetExecutingAssembly()** взима асемблито, от което е стартирания код (в случая, кода от примера **Demo-3-AssemblyTypesInfo**).

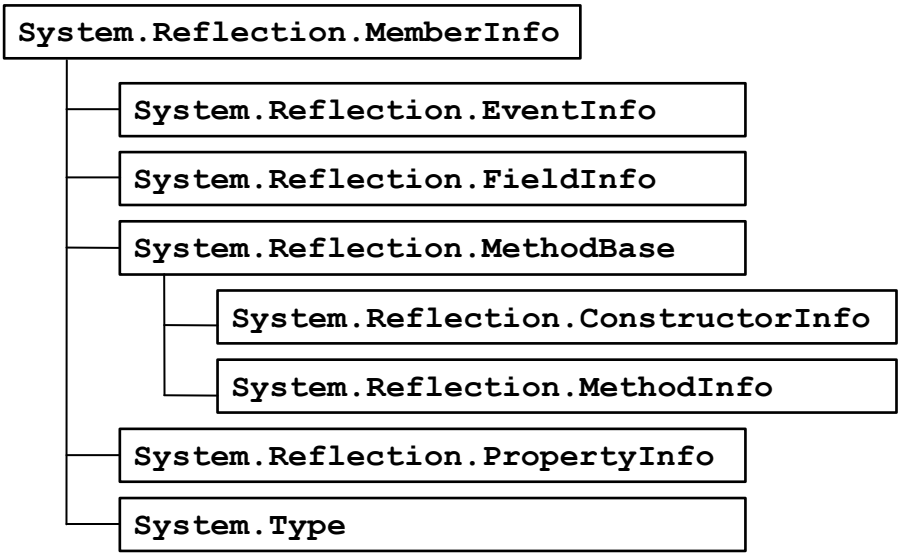
След като сме получили стартираното асембли, с помощта на метода **Assembly.GetTypes()** извличаме всички типове от него и получаваме масив от **System.Type** обекти. Всеки елемент от масива представлява отражение на един тип, дефиниран в нашето асембли.

За всеки от получените **System.Type** обекти викаме метода **System.Type.FindMembers(…)**. Параметрите, които подаваме на този метод, определят какви видове членове искаме да намерим. Първият параметър (**MemberTypes.Method**) на този метод указва, че искаме да търсим методите в дадения тип. С останалите параметри указваме как методът **System.Type.FindMembers(…)** да извърши търсенето. Вторият параметър указва, че искаме да търсим инстанции на непублични методи на дадения тип. Третият параметър указва как да се филтрират намерените методи, в случая по име. Четвъртият параметър указва самия филтър, по който да се извършва филтрирането на намерените методи на типа.

Методът **System.Type.FindMembers(…)** връща отраженията на намерените методи в даден тип. За всеки метод се отпечатват името му и в кой тип е деклариран.

Reflection класове за видовете членове

В .NET Framework е реализиран класът **System.Reflection.MemberInfo**, пряк наследник на **System.Object**. Класът **MemberInfo** е базов абстрактен клас за всички класове, позволяващи извличане на информация за членовете на даден тип с помощта на отражение. Следващата фигура показва йерархията на класовете наследници на **MemberInfo**:



Както се вижда от фигурата, за всеки вид член на тип има съответен клас (отражение), който го описва. Класовете в FCL **EventInfo**, **FieldInfo**, **MethodInfo**, **ConstructorInfo**, **PropertyInfo** и **Type** позволяват достъп съответно до метаданните на събития, полета, методи, конструктори и вложени типове, които се съдържат в даден тип.

При извличането на членовете от даден тип с помощта на метода **GetMembers(…)** на класа **System.Type** се връща масив от обекти от тип

MemberInfo. Всички останали методи за достъп до членовете на даден тип на класа **Type**, като **GetConstructors(…)**, **GetEvents(…)**, **GetFields(…)**, **GetMethods(…)**, **GetProperties(…)** и **GetNestedType(…)**, връщат съответно масив от инстанции на типовете **ConstructorInfo**, **EventInfo**, **FieldInfo**, **MethodInfo**, **PropertyInfo** и **Type**.

Извличане на методи и параметрите им

Следващият фрагмент от код показва как като имаме инстанция на някакъв наш тип, можем да получим отражението на даден метод и да извлечем параметрите му.

```
MethodInfo someMethod = myType.GetMethod("SomeMethod");
foreach (ParameterInfo param in someMethod.GetParameters())
{
   Console.WriteLine(param.ParameterType);
}
```

В показания код, първо с помощта на метода **Type.GetMethod(…)** получаваме отражение на метода с име "**SomeMethod**". След това извикваме метода **MethodInfo.GetParameters(…)** и получаваме параметрите на метода в масив от тип **ParameterInfo**. В конзолата, за всеки извлечен параметър се извежда неговият тип.

Динамично инстанциране на тип

В някои приложения се налага не само да получим информация за даден тип, дефиниран в дадено асемблы, но и да създадем инстанция на този тип. Един от начините за това е с помощта на класа **System.Activator**. Този клас се използва за динамично създаване или активиране на даден тип. Класът **System.Activator** предлага следните статични методи за създаване на инстанции на даден тип:

- **CreateInstance(…)** – създава инстанция на посочен тип подаден като **string** обект или като инстанция на **System.Type**.
- **CreateInstanceFrom(…)** – инстанцира определен тип от дадено асемблы. Името на асемблито и типа се подават като символни низове.
- **CreateComInstanceFrom(…)** – създава инстанция на COM обект. Името на типа и на файла където е дефиниран той, се подават като символни низове.

Динамично извикване на членове на даден тип

Динамичното извикване на даден член на даден тип се извършва на две стъпки. На първата стъпка се избира подходящият член, който искаме да бъде извикан. Тази стъпка се нарича свързване (binding). На втората

стъпка се активира намереният член. Тази стъпка се нарича извикване (invoking).

Един от начините да се свържем с даден член на даден тип, без той да бъде извикан, е с помощта на методите **`Type.GetConstructors(…)`**, **`Type.GetFields(…)`**, **`Type.GetMethods(…)`**, **`Type.GetProperties(…)`**. Всички тези методи връщат референции към обекти, чиито тип предлага следните методи за достъп до специфичен член на типа:

- **`FieldInfo.GetValue(…)`** – взима стойност на поле.
- **`FieldInfo.SetValue(…)`** – задава стойност на поле.
- **`ConstructorInfo.Invoke(…)`** – извиква конструктор и създава инстанция на типа.
- **`PropertyInfo.GetValue(…)`** – извиква метода за извличане на свойство **`get`**.
- **`PropertyInfo.SetValue(…)`** – извиква метода за установяване на свойство **`set`**.
- **`MethodInfo.Invoke(…)`** – извиква метод на тип.

Друг начин за динамично извикване на членовете на даден тип е с помощта на метода **`Type.InvokeMember(…)`**, който ще разгледаме след малко.

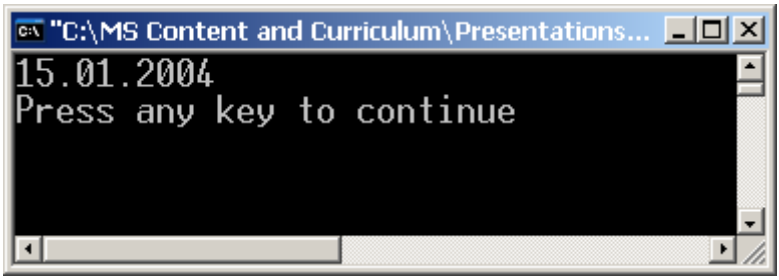
Динамично извикване на методи – пример

В настоящия пример се демонстрира динамично създаване на инстанция от тип **`System.DateTime`**, извикването на неин метод и прочитане на нейно свойство.

```
using System;
using System.Reflection;

class LateBindingDemo
{
  static void Main()
  {
    // Load the assembly mscorlib.dll
    Assembly mscorlibAssembly = Assembly.Load("mscorlib.dll");

    // Create an instance of DateTime by calling
    // new DateTime(2004, 1, 5)
    Type systemDateTimeType =
      mscorlibAssembly.GetType("System.DateTime");
    object[] constructorParams = new object[] {2004, 1, 5};
    object dateTimeInstance = Activator.CreateInstance(
      systemDateTimeType, constructorParams);

    // Invoke DateTime.AddDays(10)
```

```
      Type[] addDaysParamsTypes =
        new Type[] {typeof(System.Double)};
      MethodInfo addDaysMethod = systemDateTimeType.GetMethod(
        "AddDays", addDaysParamsTypes);
      object[] addDaysParams = new object[] {10};
      object newDateTimeInstance =
        addDaysMethod.Invoke(dateTimeInstance, addDaysParams);

      // Get the value of the property DateTime.Date and print it
      PropertyInfo datePropertyInfo =
        systemDateTimeType.GetProperty("Date");
      object datePropertyValue =
        datePropertyInfo.GetValue(newDateTimeInstance, null);
      Console.WriteLine("{0:dd.MM.yyyy}", datePropertyValue);
    }
}
```

След изпълнение на примера се получава следният резултат:



Описание на примера

Първоначално се зарежда асемблито **mscorlib.dll**, което съдържа повечето системни типове от пространството **System** на .NET Framework.

Чрез извикване на **Activator.CreateInstance(…)** се създава инстанция на типа **System.DateTime**, като на конструктора на **System.DateTime** се подават като параметри три целочислени стойности, представящи датата 05.01.2004.

След това с помощта на метода **Type.Invoke(…)** се извлича отражението на метода **AddDays(double)** на типа **System.DateTime.** Така извлеченият метод се извиква с параметър **10**, който добавя 10 дни към датата 05.01.2004. В резултат се връща обект, който е инстанция на типа **System.DateTime.**

От получения **System.DateTime** обект се извлича и отпечатва стойността на свойството с име **Date**. Отпечатаната дата е 15.01.2004.

Методът Type.InvokeMember(...)

При извикване на метода **Type.InvokeMember(…)** вътрешно се извършва свързването и извикването на търсения член на дадения тип.

Когато викаме метода **Type.InvokeMember(…)** за извикване на метод на даден тип и този метод не бъде намерен, се подава изключение **System.MissingMethodException**. В случай, че методът бъде намерен, той се извиква и **InvokeMember(…)** връща резултата от извикването на намерения метод. Методът **InvokeMember(…)** връща **null** ако извиканият метод е дефиниран като **void**.

В класа **Type** има няколко предефинирани версии на **InvokeMember(…)**. Ще разгледаме версията с най-много параметри. Другите предефинирани версии на **InvokeMember(…)** приемат някои от параметрите с подразбиращи се стойности. Методът има следната дефиниция:

```
public object InvokeMember(
   string name,
   BindingFlags invokeAttr,
   Binder binder,
   object target,
   object[] args,
   CultureInfo culture
);
```

Всички параметри, с изключение на **target**, подавани на **InvokeMember(…)** указват с какъв член на даден тип той трябва да се свърже.

Параметърът **name** указва името на члена, с който искаме да се свърже **InvokeMember(…)**.

Параметърът **invokeAttr** от тип **System.Reflection.BindingFlags** указва правилата, по които **InvokeMember(…)** трябва да избере само един член от даден тип.

Параметърът **binder** от тип **System.Reflection.Binder** указва типовете на параметрите, които **InvokeMember(…)** трябва да използва, за да извика даден метод.

Параметърът **target** е референция към обект, чийто метод искаме да извикаме с метода **InvokeMember(…)**.

Параметърът **args** от тип **System.Object** подава на **InvokeMember(…)** аргументите, с които да бъде извикан извлечения от **InvokeMember(…)** метод.

Параметърът **culture** се използва при свързване на **InvokeMember(…)** с даден метод. **InvokeMember(…)** използва дадена култура, подадена с този параметър, при конвертиране на типовете на аргументите за метод, който искаме да извикаме.

Всеки път, когато се вика методът `InvokeMember(…)`, той извършва свързване с конкретен член на даден тип и след това го извиква. Процесът на свързване отнема време. В случай, че често се осъществява достъп до даден член на даден тип, по-добрият подход е следният – един път да се свържем с него, с помощта на някои от методите, предоставени от класа `System.Type`, след което да го извикваме колкото пъти желаем.

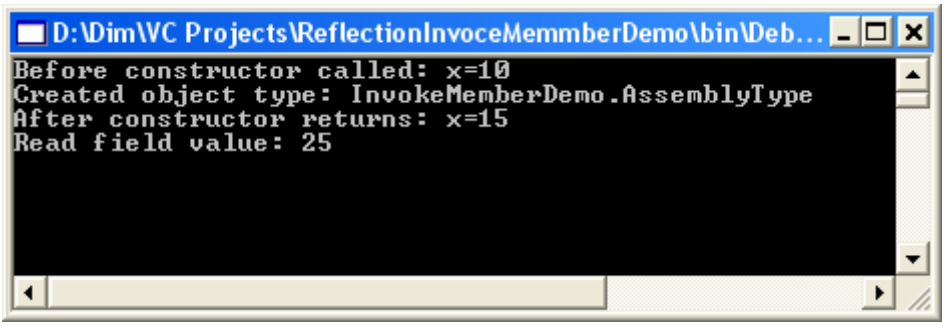
InvokeMember(…) – пример

Със следващия кратък пример демонстрираме използването на метода **`InvokeMember(…)`**:

```
using System;
using System.Reflection;

namespace InvokeMemberDemo
{
   class AssemblyType
   {
      int mAssemblyField;

      public AssemblyType(ref int x)
      {
         x = x + 5;
         mAssemblyField  = x + 10;
      }
   }

   class InvokeMemeberDemo
   {
      static void Main()
      {
         // Get AssemblyType type
         Type t = Type.GetType("InvokeMemberDemo.AssemblyType");
         object[ ] args = new Object[] { 10 };
         Console.WriteLine("Before constructor called: x={0}",
            args[0]);
         BindingFlags bf = BindingFlags.Public |
            BindingFlags.NonPublic | BindingFlags.Instance;

         // Create AssemblyType object
         Object obj = t.InvokeMember(null, bf |
            BindingFlags.CreateInstance, null, null, args, null);
         Console.WriteLine("Created object type: {0}",
            obj.GetType());
         Console.WriteLine("After constructor returns: x={0}",
            args[0]);
```

```
        // Read object field
        int value = (int) t.InvokeMember("mAssemblyField", bf |
          BindingFlags.GetField, null, obj, null, null);
        Console.WriteLine("Read field value: {0}", value);
      }
   }
}
```

След изпълнение на програмата получаваме следния резултат:



Как работи примерът?

Примерът демонстрира динамично създаване на инстанция от тип **AssemblyType** и прочитане на неговото поле **mAssemblyField** с помощта на метода **Type.InvokeMember(…)**.

В примера е дефиниран клас **AssemblyType**, в който има конструктор **AssemblyType(ref int x)** и **int** поле **mAssemblyField**.

Първоначално в примера получаваме **System.Type** обект, представящ дефинирания от нас **AssemblyType** тип.

В примера с помощта на метода **Type.InvokeMember(…)** динамично извикваме конструктора **AssemblyType(ref int x)** на дефинирания от нас тип.

Методът **Type.InvokeMember(…)** се извиква със следните параметри, за да може да се свърже с конструктора на **AssemblyType** типа и да го извика:

- **null** – указва името на члена, който искаме да извикаме с помощта на метода **Type.InvokeMember(…)**.
- **bf|BindingFlags.CreateInstance** - флагът **CreateInstance** подаден на метода **Type.InvokeMember(…)**, посочва, че **Type.InvokeMember(…)** трябва да се свърже и извика конструктор. Параметърът **bf** указва, че конструкторът, който искаме да извикаме, трябва да се търси както между публичните (**BindingFlags.Public**), така и между непубличните (**BindingFlags.NonPublic**) членове, които не са статични (**BindingFlags.Instance**).

- **null** – указва типовете на параметрите, които трябва да подадем на конструктора, който искаме да извикаме с помощта на метода **Type.InvokeMember(…)**.
- **null** – указва обекта, чиито член искаме да извикаме с помощта на метода **Type.InvokeMember(…)**.
- **args** – съдържа параметрите, с които трябва да се извиква конструктора на **AssemblyType** типа, извикван с помощта на метода **Type.InvokeMember(…)**.
- **null** – указва културата, която трябва да се използва при свързване на конструктора с метода **Type.InvokeMember(…)**.

Резултатът от извикването на метода **Type.InvokeMember(…)** с посочените параметри е обект от **AssemblyType** тип. Типът на новосъздадения обект, след извикване на метода **Type.InvokeMember(…)** се отпечатва на конзолата.

При извикване на конструктора **AssemblyType(ref int x)** от метода **Type.InvokeMember(…)**, се инициализира полето **mAssemblyField**, дефинирано в типа **AssemblyType** и стойността на подадения аргумент на конструктора **AssemblyType(ref int x)** се увеличава с 5.

Стойността на аргумента, подаден на конструктора на **AssemblyType** типа, след извикването му от метода **Type.InvokeMember(…)**, се отпечатва на конзолата.

След като сме създали обект от тип **AssemblyType**, викаме повторно метода **Type.InvokeMember(…)** с цел да извлечем стойността на неговото **mAssemblyField** поле.

Параметрите, които подаваме при второто извикване на **InvokeMember(…)**, са следните:

- **"mAssemblyField"** – указва името на полето чиято стойност искаме да прочетем.
- **bf | BindingFlags.GetField** – указва, че искаме да получим стойността на полето **mAssemblyField**
- **null** – указва типовете на параметрите, които трябва да подадем на метода, който искаме да извикаме с помощта на **InvokeMember(…)**.
- **obj** – съдържа обекта от тип **AssemblyType**, стойността на чието поле искаме да прочетем.
- **null** – указва, параметрите които трябва да подадем на метода, който искаме да извикаме с помощта на **InvokeMember(…)**.
- **null** – указва културата, която трябва да се използва при свързване на метода **Type.InvokeMember(…)** с търсения член.

Резултатът от извикването на **Type.InvokeMember(…)** с горепосочените параметри е стойността на полето **mAssemblyField** в динамично създадения от нас обект от тип **AssemblyType**. Получената стойност се отпечатва на конзолата.

Reflection Emit

System.Reflection.Emit е пространство от имена, предоставящо класове, с чиято помощ компилатори и приложения могат да създават нови асемблита, типове, методи и да генерират динамично **Microsoft Intermediate Language (MSIL)** инструкции. Класовете от това пространство намират голямо приложение при разработка на компилатори и интерпретатори за скриптови езици. С класовете от това пространство могат да се създават цели асемблита, да се изпълняват и да се запазват на диска.

Използване на Reflection Emit

Класовете от **System.Reflection.Emit** позволява създаването, както на цяло асембли, така и на отделни негови модули. Създаването на типове в даден модул, по време на изпълнение, дефинирането на методи, събития и свойства също налага използването на класовете от пространството от имена **System.Reflection.Emit**. Класовете, предоставящи тези възможности, са:

- **AssemblyBuilder** – клас, позволяващ динамично създаване на асембли. Този клас е наследник на класа **System.Assembly**. Дефинираните в него методи позволяват зареждане и създаване на модули, дефиниране на ресурси, както и записване на динамично създадено асембли във файл. С метода **AssemblyBuilder.SetEntryPoint(…)** се задава входна точка за изпълнение на дадено асембли.
- **ModuleBuilder** – клас, позволяващ дефиниране и динамично създаване на модули за дадено асембли.
- **TypeBuilder** – клас, позволяващ дефиниране и динамично създаване на типове. Този клас е наследник на класа **System.Type**. Дефинираните в него методи позволяват създаването на всички видове членове за даден тип.
- **ConstructorBuilder** – клас, позволяващ динамично създаване на конструктори за даден тип. Този клас е наследник на класа **System.ConstructorInfo**.
- **MethodBuilder** – клас, позволяващ динамично създаване на методи за даден тип и предоставящ методи и свойства за работа с тях. Този клас е наследник на класа **System.MethodInfo**.
- **PropertyBuilder** – клас, позволяващ динамично създаване на свойства за даден тип и предоставящ методи и свойства за работа с тях. Този клас е наследник на класа **System.PropertyInfo**.

- **EventBuilder** – клас, позволяващ дефинирането на събития за даден клас.

Генериране на MSIL инструкции

Динамичното генериране на MSIL инструкции става с помощта на класа **System.Reflection.Emit.ILGenerator**. Използването на този клас позволява по време на изпълнение на дадена програма да се добавят MSIL инструкции за даден метод или конструктор. Класът **ILGenerator** предоставя методите **Emit(…)** и **EmitCall(…)** за добавяне на последователност от MSIL инструкции. С помощта на други методи на този клас могат да се декларират локални променливи и да се създават блокове за прихващане на изключения в даден метод. Методът **EmitWriteLine(…)** добавя инструкции за отпечатване на низ на конзолата.

Класовете **MethodBuilder** и **ConstructorBuilder** предоставят метод **GetILGenerator()**, който връща съответно **ILGenerator** за метод или конструктор.

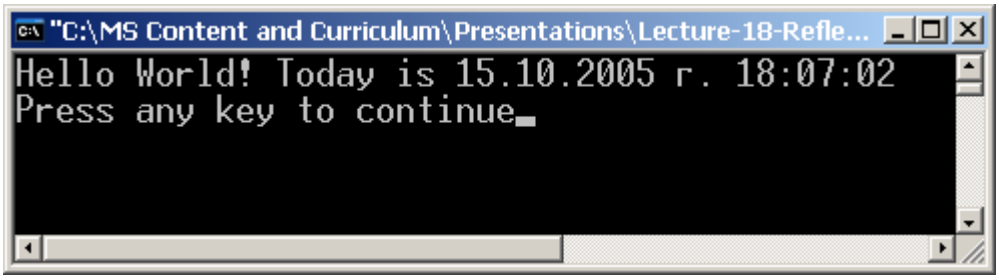
Динамично генериране на асембли – пример

В настоящия пример се създава асембли по време на изпълнение на програмата. Дефинира се модул, тип и метод. Създаденото асембли се записва във файл. Следващият код демонстрира как става това:

```
using System;
using System.Reflection;
using System.Reflection.Emit;

class ReflectionEmitDemo
{
  static void Main()
  {
    AssemblyName assemblyName = new AssemblyName();
    assemblyName.Name = "DynamicAssembly";

    // Create new assembly
    AssemblyBuilder newAssembly =
      AppDomain.CurrentDomain.DefineDynamicAssembly(
        assemblyName, AssemblyBuilderAccess.RunAndSave);

    // Create new module in the new assembly
    ModuleBuilder newModule = newAssembly.DefineDynamicModule(
      "NewModule", "EmitedAssembly.exe");

    // Create new type in the new module
    TypeBuilder newType = newModule.DefineType(
      "HelloWorldType", TypeAttributes.Public);

    // Create new method in the new type
```

```
        MethodBuilder newMethod = newType.DefineMethod(
          "WriteHello", MethodAttributes.Static |
          MethodAttributes.Public, null, null);

        // Generate the MSIL code in the new method
        ILGenerator msilGen = newMethod.GetILGenerator();
        msilGen.EmitWriteLine("Hello World! Today is " +
          DateTime.Now);
        msilGen.Emit(OpCodes.Ret);

        // Instantiate the new type
        Type helloWorldType = newType.CreateType();
        Object instance = Activator.CreateInstance(helloWorldType);

        // Run the method WriteHello from the new type
        MethodInfo helloWorldMethod =
          helloWorldType.GetMethod("WriteHello");
        helloWorldMethod.Invoke(instance, null);

        // Save the assembly to an executable file
        newAssembly.SetEntryPoint(helloWorldMethod);
        newAssembly.Save("EmitedAssembly.exe");
    }
}
```

След изпълнение на примера се получава следният резултат:



Описание на примера

Първоначално в примера се създава ново асембли, с помощта на метода **`AppDomain.CurrentDomain.DefineDynamicAssembly(…)`**. Този метод приема два параметъра - името на асемблито, което искаме да създадем, и флаг, указващ, че създаденото асембли трябва да може да се изпълнява и записва във файл.

В новосъздаденото асембли чрез извикване на метода **`AssemblyBuilder.DefineDynamicModule(…)`** се създава нов модул с име **`EmitedAssembly.exe`**, след което към него се създава нов публичен тип **`HelloWorldType`**. Създаденото в нашия пример асембли се състои от само един модул.

В типа **HelloWorldType** се създава публичен статичен метод **WriteHello()**. При създаването на метода се получава обект от тип **MethodBuilder**, от който се взима обект от тип **ILGenerator** за новосъздадения метод.

В метода **WriteHello()** се генерира последователност от MSIL инструкции, която печата текст на конзолата.

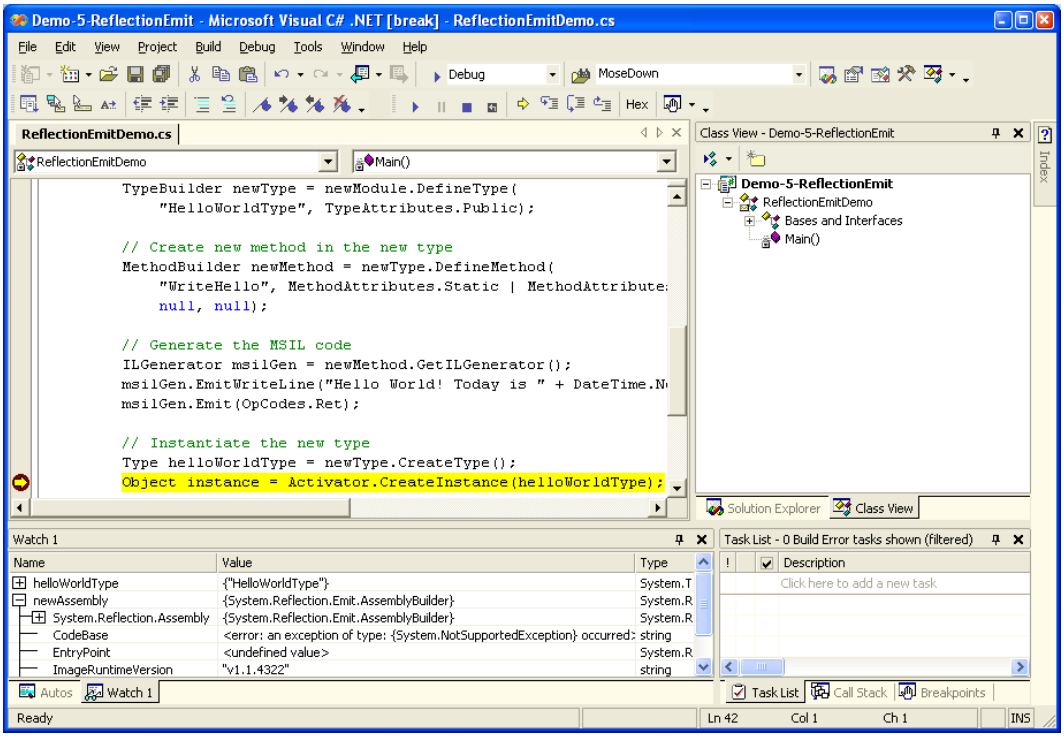
За да се демонстрира, че работи, новосъздаденият тип се инстанцира и му се извиква метода **WriteHello()**, който е дефиниран като статичен.

За входна точка на генерираното асембли се задава методът **WriteHello()** и след това асемблито се записва във файл **EmitedAssembly.exe**.

Проследяване на изпълнението на примера

За проследяване стъпка по стъпка изпълнението на примера, можем да използваме проекта **Demo-5-ReflectionEmit** от демонстрациите.

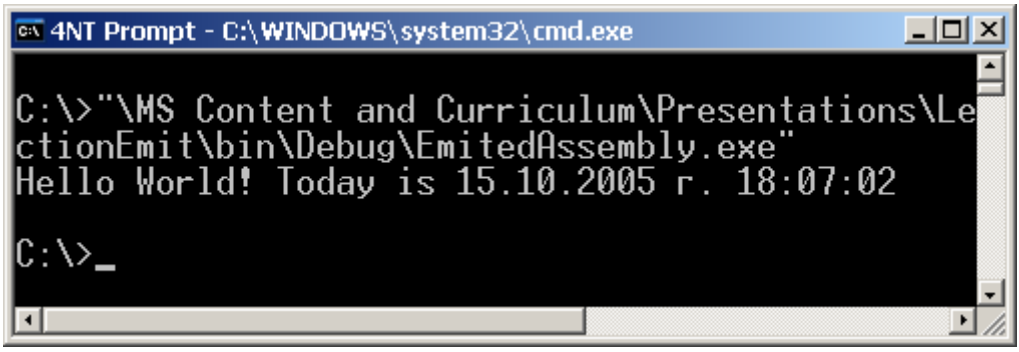
1. Отваряме проекта **Demo-5-ReflectionEmit.sln**, който съдържа кода от горния пример.
2. Слагаме точка на прекъсване на последния ред на **Main()** метода.
3. Стартираме приложението с **[F5]**.



На картинката е показан изглед от VS.NET в момент на изпълнение на примера.

4. След като сме изпълнили примера отиваме в директорията на проекта, отваряме поддиректорията **.\bin\Debug** и виждаме, че е създаден файл с име **EmitedAssembly.exe**.

5. Стартираме от конзолата **EmitedAssembly.exe** и получаваме следния резултат:



Както се вижда, генерираното асембли е напълно функционално и може да се изпълни както всеки друг **.exe** файл.

Упражнения

1. Какво е Global Assembly Cache? За какво служи?
2. Опишете поне един начин за преглеждане на асемблитата от Global Assembly Cache.
3. Да се реализира Windows Forms приложение, което позволява да се зарежда избрано от потребителя асембли и показва информация за него (път от където е заредено, дали е заредено от GAC, входната му точка и т.н.).
4. Да се реализира конзолно приложение, което зарежда асемблито **mscorlib.dll** и отпечатва имената на всички типове в него.
5. Да се реализира конзолно приложение, което зарежда асемблито **mscorlib.dll** и намира всички методи на типа **System.DateTime**, който е дефиниран в него.
6. Съставете Windows Forms приложение, което зарежда асембли, името на което се избира от потребителя, и извлича от него имената и параметрите на конструкторите на всички типове, дефинирани в него.
7. Дефинирайте интерфейс **ICalculatable**, който дефинира метод **double Calculate(int[])**. Напишете конзолно приложение, което чете от текстов файл редица от числа, намира всички асемблита от зададена директория, в които има имплементация на **ICalculatable** и чрез всяко от тях извършва пресмятането **Calculate(…)** и отпечатва резултата. Тествайте като създадете две асемблита, в които има тип,

имплементиращ **`ICalculatable`**. Едното асембли трябва да изчислява средно аритметично, а другото сума на елементите от подадения масив.

8. Съставете програма, която прочита въведена текстова последователност и създава асембли съдържащо тип, който съдържа метод отпечатващ тази текстова последователност. Генерираното асембли трябва да бъде съхранено, като изпълним файл.

Използвана литература

1. Ивайло Христов, Отражение на типовете (Reflection) – http://www.nakov.com/dotnet/lectures/Lecture-18-Reflection-v1.0.ppt
2. Георги Иванов, Отражение на типовете (Reflection) – http://www.nakov.com/dotnet/2003/lectures/Reflection.doc
3. Jeffrey Richter, Applied Microsoft .NET Framework Programming, Microsoft Press, 2002, ISBN 0735614229
4. Jesse Liberty, Programming C#, 3rd Edition, O'Reilly, 2003, ISBN 0596004893
5. Professional C#, 3rd, Wrox Press, 2004, ISBN 0764557599
6. MSDN Library – http://msdn.microsoft.com

Глава 20. Сериализация на данни

Автор

Радослав Иванов

Необходими знания

- Базови познания за .NET Framework, CLR (Common Language Runtime) и общата система от типове в .NET (Common Type System)
- Познания за езика C#
- Познания за работа с потоци от данни
- Познания по отражение на типовете (reflection)
- Познания за атрибутите в .NET Framework
- Познания за работа с XML в .NET Framework

Съдържание

- Какво е сериализация? Кога и защо се използва?
- Форматери (Formatters)
- Процесът на сериализация
- Сериализация и десериализация – пример
- Пример за бинарна сериализация
- Пример за сериализация по мрежата
- Пример за дълбоко копиране на обекти
- **`IDeserializationCallback`**
- Контролиране на сериализацията. **`ISerializable`**
- XML сериализация
- Контролиране на изходния XML

В тази тема ...

В настоящата тема ще разгледаме сериализацията на данни в .NET Framework. Ще обясним какво е сериализация, за какво се използва и как да контролираме процеса на сериализация. Ще се запознаем с видовете форматери (formatters). Ще обясним какво е XML сериализация, как работи тя и как можем да контролираме изходния XML при нейното използване.

Сериализация

В съвременното програмиране често се налага да се съхрани състоянието на даден обект от паметта и да се възстанови след известно време. Това позволява обектите временно да се съхраняват на твърдия диск и да се използват след време, както и да се пренасят по мрежата и да се възстановяват на отдалечена машина.

Проблемите при съхранението и възстановяването на обекти са много и за справянето с тях има различни подходи. За да се намалят усилията на разработчиците в .NET Framework е изградена технология за автоматизация на този процес, наречена **сериализация**. Нека се запознаем по-подробно с нея.

Какво е сериализация (serialization)?

Сериализацията е процес, който преобразува обект или свързан граф от обекти до поток от байтове, като запазва състоянието на неговите полета и свойства. Потокът може да бъде двоичен (binary) или текстов (XML).

Какво е десериализация (deserialization)?

Обратният процес на сериализацията е десериализацията. Десериализацията е процеса на преобразуване на поток от байтове обратно до обект. Десериализираният (възстановеният) обект запазва състоянието на оригиналния обект (стойностите в полетата и свойствата си).

Кога се използва сериализация?

Ще разгледаме някои от най-честите приложения на сериализацията и десериализацията.

Запазване на състоянието на обект

Сериализацията се използва за съхранение на информация и запазване на състоянието на обекти. Използвайки сериализация, дадена програма може да съхрани състоянието си във файл, база данни или друг носител и след време да го възстанови обратно.

Предаване на обект през комуникационна мрежа

Сериализацията може да се използва за предаване на обекти през мрежа. За целта обектът се сериализира и се транспортира през мрежата, след което се десериализира, за да се пресъздаде абсолютно същия обект, който е бил изпратен. Примерно приложение на този метод е за предаване на данни между две програми.

Приложение вътрешно в .NET Framework

Технологиите от .NET Framework използват вътрешно сериализация за някои задачи, например:

- за запазване на състоянието на сесията (т. нар. "session state") в ASP.NET
- за копиране на обекти в clipboard в Windows Forms
- за предаване на обекти по стойност от един домейн на приложение (application domain) в друг
- за дълбоко копиране на обекти (deep copy)
- в технологията за отдалечено извикване .NET remoting

Други приложения

След като един обект бъде превърнат в поток от байтове, той може да бъде криптиран, компресиран или обработен по друг начин в съответствие с целта, която сме си поставили. Тези процеси са прозрачни, т.е. не зависят от сериализирания обект. Обектът се сериализира и ние обработваме потока от байтове, без да се интересуваме какви са структурата и съдържанието на обекта. Така сериализацията улеснява обработката на обекти понеже позволява да се запишат в поток.

Защо да използваме сериализация?

Запазването на един обект може да се направи и ръчно, без използването на сериализация. Този подход често е трудоемък и предразполага към допускане на много грешки. Процесът става по-сложен, когато се налага да запазим йерархия от обекти.

Представете си, че изграждате бизнес приложение с 10 000 класа и трябва да запазите сложен граф от навързани един с друг обекти. Представете си как се налага да пишете код във всеки клас, който се справя с протоколи, несъвпадение на типовете при клиент/сървър, управление на грешки, обекти сочещи към други обекти (циклично), работа със структури, масиви и т.н. При по-старите платформи се работеше така, защото нямаше автоматична сериализация.

Сериализацията в .NET е автоматична

Сериализацията в .NET Framework прави целия този процес по обхождането на графа, започващ от даден обект и записването му в поток прозрачен и автоматичен. Тя ни дава удобен механизъм за реализирането на такава функционалност с минимални усилия.

Сериализиране на циклични графи от обекти

С помощта на сериализацията можем да сериализираме циклични графи от обекти, т.е. обекти, които се реферират едни от други. В общия случай съхраняването и предаването на такива структури не е лесно, но в .NET

Framework това се реализира от CLR и грижата не е на програмиста. Форматерът сериализира всеки обект само по веднъж и не влиза в безкраен цикъл (форматерите ще обсъдим малко по-нататък в тази тема).

Кратък пример за сериализация?

Следващият фрагмент код илюстрира как можем да сериализираме обект и да го запишем в бинарен файл със средствата на .NET Framework:

```
string str = ".NET Framework";
BinaryFormatter f = new BinaryFormatter();
using (Stream s = new FileStream("sample.bin", FileMode.Create))
{
   f.Serialize(s, str);
}
```

На първия ред е дефиниран обектът, който ще сериализираме. Той може да бъде всякакъв тип – **Int32**, **String**, **DateTime**, **Exception**, **Image**, **ArrayList**, **HashTable**, потребителски дефиниран клас и т.н. В случая сме използвали обект от тип **string**. Обектът, който ще бъде сериализиран, трябва да отговаря на специални изисквания, които ще обясним по-нататък в настоящата тема.

За да сериализираме обект, трябва да създадем форматер (formatter). Форматерът е специален клас, който имплементира интерфейса **IFormatter**. Той извършва цялата работа по сериализирането и десериализирането на йерархия (граф) от обекти и записването им в поток. Сериализирането се извършва от метода **Serialize(…)**. Като първи параметър, този метод очаква наследник на класа **System.IO.Stream**. Това е потокът, в който ще се сериализират данните, което означава, че обектът може да се сериализира в **MemoryStream**, **FileStream**, **NetworkStream** и т.н. Вторият параметър на метода е обектът, който ще се сериализира.

Потокът, в който ще сериализираме обекта е дефиниран на третия ред в примерния фрагмент код. Използваната **using** конструкция гарантира затварянето на използвания в нея поток след приключване на работата с него.

Сериализацията на обекта се извършва чрез извикване на метода **Serialize(…)**. В процеса на сериализация се обхождат (чрез **reflection**) всички член-променливи на обекта и се сериализират само членовете на инстанцията, без статичните й членове. Видимостта на член-променливата няма значение – сериализират се дори **private** полетата.

Форматери (Formatters)

Форматерите съдържат логиката за записване на резултата от сериализацията в поток, т.е. реализират форматираща логика. Форматерът е клас, който имплементира интерфейса **IFormater**. Методът му **Serialize(…)**

преобразува обекта до поток от байтове. Методът **`Deserialize(…)`** чете данните от потока и пресъздава обекта.

Форматерите съдържат логиката за форматиране на сериализираните обекти. CLR обхожда метаданните за член-променливи и чрез reflection извлича стойностите им. Извлечените стойностите се подават след това на форматера, за да ги запише по подходящ начин в потока.

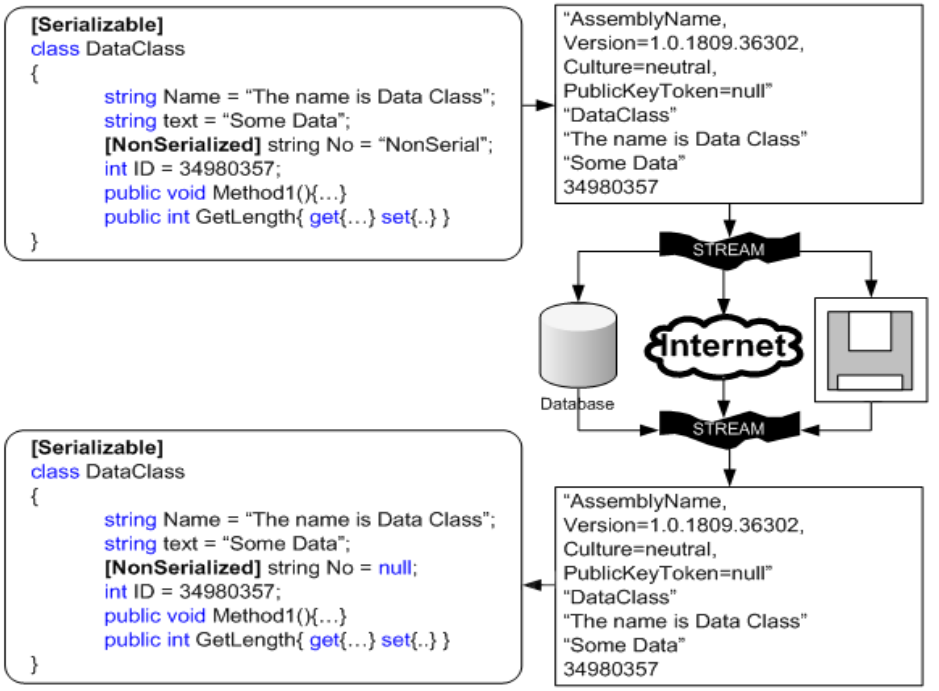
.NET Framework ни осигурява два стандартни форматера, дефинирани в пространството **`System.Runtime.Serialization`**:

- **`BinaryFormatter`** – сериализира обект в двоичен формат. Полученият в резултат на сериализацията поток е много компактен.
- **`SoapFormatter`** – сериализира обект в SOAP формат. За разлика от двоичния формат, SOAP форматът осигурява съвместимост с други системи, защото представлява XML-базиран стандарт за обмяна на съобщения и е независим от платформата. SOAP стандартът ще разгледаме в детайли в [темата за уеб услуги](#).

Можем да създаваме потребителски дефинирани форматери. Те наследяват абстрактния клас **`Formatter`**, осигуряващ базова функционалност.

Процесът на сериализиране

На фигурата схематично е показано как работят процесите на сериализиране и десериализиране в .NET Framework:



При сериализирането на обекта в потока се записват името на класа, името на асемблито (assembly) и друга информация за обекта, както и всички член-променливи, които не са маркирани като **[NonSerialized]** (употребата на този атрибут ще обясним по-нататък в тази тема). При десериализацията информацията се чете от потока и се пресъздава обектът.

Кратък пример за сериализация

Настоящият пример илюстрира сериализирането на обекти, като се обръща внимание на някои изисквания, на които трябва да отговаря сериализираният обект:

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

[Serializable]
class FirstExample
{
   public int mNumber;
   [NonSerialized] public int mId;
   public string mName;
}

class Serializer
{
   public void Serialize()
   {
      FirstExample obj = new FirstExample();
      BinaryFormatter f = new BinaryFormatter();
      using (Stream stream = new FileStream(
         "x.bin", FileMode.Create))
      {
         f.Serialize(stream, obj);
      }
   }

   public void Deserialize() {...}
}
```

Как работи примерът?

Нека разгледаме класа **FirstExample**, който сме дефинирали в примера. Обърнете внимание на атрибута **[Serializable]**, намиращ се преди дефиницията на класа. Приложен към даден тип, този атрибут указва, че инстанциите на типа могат да бъдат сериализирани. При опит за сериализиране на обект, чийто тип няма атрибута **[Serializable]** CLR предизвиква изключение от тип **SerializationException**. Допълнително условие, за успешната сериализация на обект е, че всички типове на член-променливите на обекта, които ще бъдат сериализирани, трябва също да притежават атрибута **[Serializable]**.

Обърнете внимание на атрибута **[NonSerialized]**, намиращ се пред декларацията на променливата **mId** в класа **FirstExample**. Чрез този атрибут указваме, че съответният член на класа не трябва да бъде сериализиран. Причините да не сериализираме някои от членовете на клас са различни – те може да съдържат секретна информация, която не трябва да бъде съхранявана или да съдържат данни, които не са нужни при пресъздаването на обекта.

Сериализация на обект от дефинирания клас **FirstExample**, ще извършим във функцията **Serialize()** на класа **Serializer**. Първо дефинираме обекта, който ще сериализираме. След това създаваме форматер, който ще извърши работата по сериализацията на обекта. В примера сме използвали форматер от тип **BinaryFormatter**, който е член на пространството **System.Runtime.Serialization.Formatters.Binary**. След създаването на форматера, създаваме потока, в който ще бъде сериализиран обекта – в примера сме използвали **FileStream**. Използваната **using** конструкция гарантира затварянето на използвания в нея поток след приключване на работата с него. Накрая извикваме функцията **Serialize(…)** на форматера и обекта се сериализира.

Кратък пример за десериализация

В този пример ще илюстрираме как протича десериализацията на обекти:

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

[Serializable]
class FirstExample
{
   public int mNumber;
   [NonSerialized] public int mId;
   public string mName;
}

class Serializer
{
   public void Serialize(){...}

   public void Deserialize()
   {
      BinaryFormatter f = new BinaryFormatter();
      using (Stream stream = new FileStream(
         "x.bin", FileMode.Open))
      {
         FirstExample fe = (FirstExample)
            f.Deserialize(stream);
      }
   }
```

```
}
```

Как работи примерът?

Този пример е логично продължение на предходния пример за сериализация. В него ще разгледаме метода **Deserialize()** на класа **Serializer**, която беше пропусната в предишния пример.

В началото на функцията **Deserialize()** създаваме форматера, който ще десериализира обекта. Отново използваме **BinaryFormatter**, понеже такъв тип форматер сме използвали при сериализирането на обекта в предишния пример. След това създаваме потока, от който ще десериализираме обекта. Накрая извикваме функцията **Deserialize(…)** на форматера, която връща като резултат десериализирания обект. Връщаният тип от функцията **Deserialize(…)** е **System.Object**, затова преди да присвоим резултата на променлива от тип **FirstExample**, трябва да го преобразуваме към този тип.

Бинарна сериализация – пример

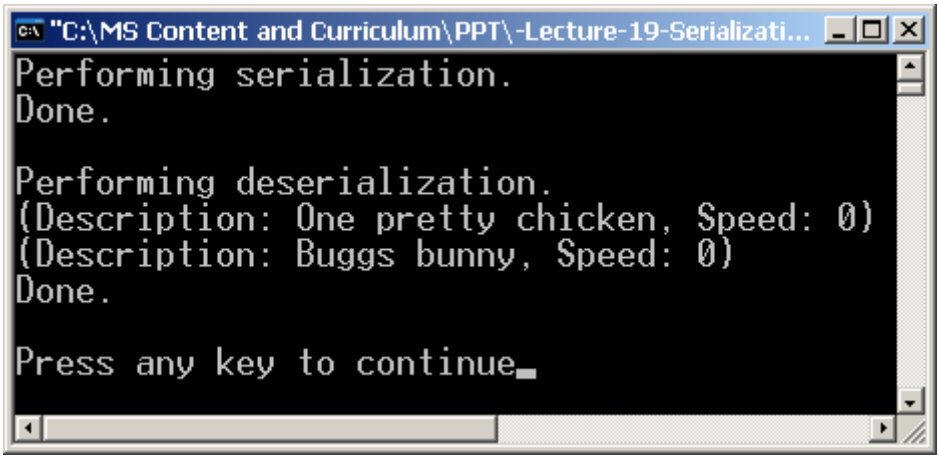
Ще представим още един пример за сериализация и десериализация на данни чрез **BinaryFormatter**:

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

[Serializable]
class Animal
{
  private string mDescription;
  [NonSerialized] private int mSpeed;

  public string Description
  {
    get
    {
      return mDescription;
    }
    set
    {
      mDescription = value;
    }
  }

  public int Speed
  {
    get
```

```
      {
         return mSpeed;
      }
      set
      {
         mSpeed = value;
      }
   }
}

class SerializeToFileDemo
{
   static void DoSerialization()
   {
      Animal animal1 = new Animal();
      animal1.Description = "One pretty chicken";
      animal1.Speed = 3;

      Animal animal2 = new Animal();
      animal2.Description = "Buggs bunny";
      animal2.Speed = 1000;

      IFormatter formatter = new BinaryFormatter();
      Stream stream =
         new FileStream("data.bin", FileMode.Create);
      using (stream)
      {
         formatter.Serialize(stream, animal1);
         formatter.Serialize(stream, animal2);
      }
   }

   static void DoDeserialization()
   {
      IFormatter formatter = new BinaryFormatter();
      Stream stream = new FileStream("data.bin", FileMode.Open);
      using (stream)
      {
         Animal animal1 = (Animal) formatter.Deserialize(stream);
         Console.WriteLine("(Description: {0}, Speed: {1})",
            animal1.Description, animal1.Speed);

         Animal animal2 = (Animal) formatter.Deserialize(stream);
         Console.WriteLine("(Description: {0}, Speed: {1})",
            animal2.Description, animal2.Speed);
      }
   }

   static void Main()
   {
```

```
      Console.WriteLine("Performing serialization.");
      DoSerialization();
      Console.WriteLine("Done.\n");

      Console.WriteLine("Performing deserialization.");
      DoDeserialization();
      Console.WriteLine("Done.\n");
   }
}
```

След изпълнение на примера, се получава следният резултат:



Как работи примерът?

В началото на примера дефинираме класа **Animal**. Атрибутът **[Serializabe]** указва, че инстанциите му могат да бъдат сериализирани. Член-променливата **mSpeed** е маркирана с атрибута **[NonSerialized]**, поради което не се сериализира.

Класът **SerializeToFileDemo** съдържа функциите **DoSerialization()** и **DoDeserialization()**, които извършват работата по сериализацията и десериализацията на обектите.

Функцията **DoSerialization()** създава две инстанции на класа **Animal**, присвоява стойности на полетата им и ги сериализира последователно в двоичен файл, като за целта използва форматер от тип **BinaryFormatter**.

Функцията **DoDeserialization()** десериализира сериализираните инстанции и отпечатва полетата им.

При стартиране на програмата се извиква метода **DoSerialization()** и след това **DoDeserialization()**, при което стойностите на полетата на сериализираните обекти се отпечатват на екрана. Забележете, че стойността на полето **Speed** се губи, защото не се сериализира заради атрибута **[NonSerialized]**, който сме използвали в класа **Animal**.

Сериализация по мрежата – пример

С настоящия пример ще онагледим как можем да сериализираме дървовидна структура от данни с **BinaryFormatter** и да я пренесем на друг компютър през TCP/IP мрежа.

В примера ще пренасяме животни (инстанции на класа **Animal**). Примерът се състои от три проекта – изпращач на данни (**AnimalSender**), получател на данни (**AnimalReceiver**) и библиотека за типовете, описващи животните (**AnimalLibrary**). Можем да ги създадем във VS.NET като три отделни проекта в едно и също решение (Solution) или като 2 решения: едното, съдържащо **AnimalSender** и **AnimalLibrary**, а другото – **AnimalReceiver** и **AnimalLibrary**. В последния случай ще имаме възможност да отворим и да дебъгваме едновременно приложенията за изпращане и за приемане на животни в отделни инстанции на VS.NET като общата част между тях (библиотеката **AnimalLibrary**) няма да се копира два пъти.

Библиотеката с типове

Библиотеката с типовете, описващи животните, е обща за изпращача и за получателя. Всички типове в библиотеката са отбелязани с атрибута **[Serializable]**, за да се позволи при нужда да бъдат сериализирани от CLR. В нея са дефинирани три типа – **Eye**, **Claws** и **Animal**:

Eye.cs

```
using System;

namespace AnimalLibrary
{
   [Serializable]
   public class Eye
   {
      private string mDescription;
      private double mDioptre;

      public Eye(string aDescription, double aDioptre)
      {
         mDescription = aDescription;
         mDioptre = aDioptre;
      }

      public override string ToString()
      {
         string result = String.Format("({0}, {1})",
            mDescription, mDioptre);
         return result;
      }
   }
}
```

Класът **Eye** съдържа две член-променливи – **mDescription** и **mDioptre**, които се инициализират от конструктора на класа. В класа е предефиниран метода **ToString()**, който връща символен низ, описващ съдържанието на обект от този тип.

Claws.cs

```
using System;

namespace AnimalLibrary
{
   [Serializable]
   public class Claws
   {
      public string mDescription;

      public Claws(string aDescription)
      {
         mDescription = aDescription;
      }

      public string Description
      {
         get
         {
            return mDescription;
         }
      }

      public override string ToString()
      {
         return mDescription;
      }
   }

}
```

Класът **Claws** съдържа една член-променлива – **mDescription**, която се инициализира от конструктора на класа. Дефинирано е свойството **Description**, което е само за четене и връща стойността на член-променливата **mDescription**. В класа е предефиниран методът **ToString()**, който връща символен низ, описващ съдържанието на обект от този тип.

Animal.cs

```
using System;
using System.Text;

namespace AnimalLibrary
{
```

```
[Serializable]
public class Animal
{
   private string mName;
   private Claws mClaws;
   private Eye[] mEyes;

   public string Name
   {
      get
      {
         return mName;
      }

      set
      {
         mName = value;
      }
   }

   public Claws Claws
   {
      get
      {
         return mClaws;
      }

      set
      {
         mClaws = value;
      }
   }

   public Eye[] Eyes
   {
      get
      {
         return mEyes;
      }

      set
      {
         mEyes = value;
      }
   }

   public override string ToString()
   {
      StringBuilder sbEyes = new StringBuilder(" ");
      foreach (Eye eye in mEyes)
```

```
        {
           sbEyes.Append(eye);
           sbEyes.Append(" ");
        }
        string eyesAsString = sbEyes.ToString();

        string result =
           String.Format("(Name: {0}, Claws: {1}, Eyes: {2})",
           mName, mClaws, eyesAsString);
        return result;
     }
   }
}
```

Класът **Animal** съдържа три член-променливи – **mName** от тип **string**, **mClaws** от тип **Claws** и **mEyes**, която е масив от тип **Eye**. В класа са дефинирани свойства за достъп до член-променливите и е предефиниран метода **ToString()**, който връща символен низ, описващ съдържанието на обект от този тип.

Защо е нужна библиотеката с типовете?

Библиотеката с типовете е нужна за да могат изпращачът и получателят да работят с един и същ, общ и за двамата, тип, който да прехвърлят през мрежата. Този тип е препоръчително да се намира в общо за двете приложения асембли. Не се препоръчва изпращачът и получателят сами да си дефинират типа, който се прехвърля.

Всъщност последното технически е възможно (от гледна точка на механизмите за сериализация на .NET Framework), но само ако класът, който се сериализира и при изпращача и при получателя е с едно и също име, от един и същ **namespace** и е дефиниран в асембли със слабо име, което и при изпращача, и при получателя има едно и също име и версия.

Препоръчително е когато се сериализират данни и двете страни (сериализиращото приложение и десериализиращото приложение) да работят с един и същ тип, т.е. да ползват общо асембли, в което е дефиниран този тип.

Приложението-изпращач на данните

Ето как изглежда сорс кодът на приложението, което изпраща инстанции на класа **Animal** по мрежата към другото приложение, което ги получава:

AnimalSender.cs

```
using System;
using System.Net.Sockets;
using System.Runtime.Serialization;
```

```
using System.Runtime.Serialization.Formatters.Binary;

using AnimalLibrary;

class AnimalSender
{
   const string SERVER_HOSTNAME = "localhost";
   const int SERVER_PORT = 10000;

   static void Main()
   {
      Animal animal = new Animal();
      animal.Name = "My fluffy cat";
      animal.Claws = new Claws("Sharp beautiful claws");
      animal.Eyes = new Eye[]
      {
         new Eye("Left eye", 1.05),
         new Eye("Right eye", 0.95)
      };

      TcpClient tcpClient =
         new TcpClient(SERVER_HOSTNAME, SERVER_PORT);
      try
      {
         IFormatter formatter = new BinaryFormatter();
         NetworkStream stream = tcpClient.GetStream();
         using (stream)
         {
            formatter.Serialize(stream, animal);
         }
         Console.WriteLine("Sent animal: {0}", animal);
      }
      finally
      {
         tcpClient.Close();
      }
   }
}
```

Приложението-изпращач създава инстанция на класа **`Animal`**, дефиниран в библиотеката **`AnimalLibrary`** и инициализира нейните полетата. След това отваря TCP сокет към получателя (чрез класа **`TcpClient`**), сериализира инстанцията и я изпраща по сокета. Счита се, че получателят слуша на порт 10 000 на локалната машина (localhost).

Приложението-получател на данните

Нека сега разгледаме и приложението, което посреща сериализираните данни и ги десериализира и използва:

AnimalReceiver.cs

```
using System.Net.Sockets;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

using AnimalLibrary;

class AnimalReceiver
{
   const int SERVER_PORT = 10000;

   static void Main()
   {
      TcpListener tcpListener =
        new TcpListener(IPAddress.Any, SERVER_PORT);
      tcpListener.Start();
      Console.WriteLine("Server started.");

      while (true)
      {
         TcpClient client = tcpListener.AcceptTcpClient();
         try
         {
            IFormatter formatter = new BinaryFormatter();
            NetworkStream stream = client.GetStream();
            using (stream)
            {
               Animal animal =
                   (Animal) formatter.Deserialize(stream);
               Console.WriteLine("Received animal: {0}", animal);
            }
         }
         finally
         {
            client.Close();
         }
      }
   }
}
```

Приложението-получател отваря сървърски TCP сокет (на порт 10 000 на локалната машина) и чака за заявки от клиента. Това се извършва с помощта на инстанция на класа **`TcpListener`**, чието предназначение е да слуша за връзки от TCP клиенти. При пристигане на заявка от клиента, приложението прочита изпратените от клиента данни и се опитва да ги десериализира в инстанция на класа **`Animal`**. След десериализацията, съдържанието на обекта се извежда в конзолата.

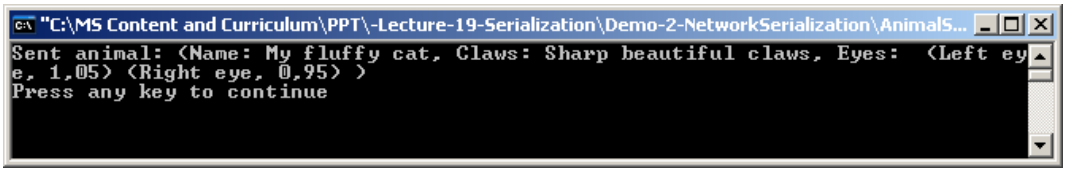
Проследяване на примера с VS.NET

За да проследим как се изпълнява примерът, можем да създадем две решения (Solutions) с VS.NET и да ги стартираме.

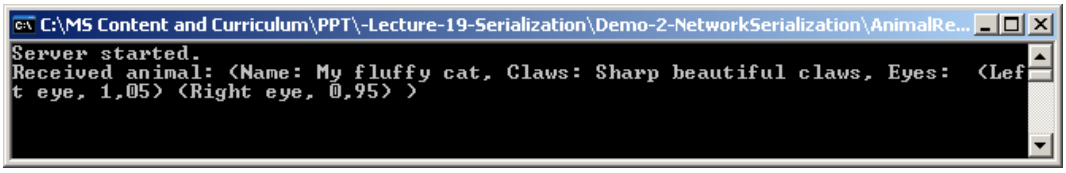
1. Стартираме VS.NET и създаваме решението **AnimalReceiver.sln**, което ще представлява сървъра (изпращача на данни). В него създаваме проектите **AnimalReceiver.csproj** и **AnimalLibrary.csproj** и копираме в тях съответния им сорс код. Стартираме сървъра с **[Ctrl-F5]**.

2. Стартираме нова инстанция на VS.NET и по същия начин създаваме решението-клиент **AnimalSender.sln**, което ще посреща изпратените данни. В него създаваме проекта **AnimalSender.csproj** и добавяме вече създадения проект **AnimalLibrary.csproj**. Копираме в проекта **AnimalSender.csproj** сорс кода от неговите класове. Стартираме клиента с **[Ctrl-F5]** и наблюдаваме прехвърлянето на данни.

При стартирането на приложението-получател, в конзолата се изписва "**Server Started.**". След стартирането на приложението-изпращач в неговата конзола се получава следният резултат:



Ако се върнем в прозореца на приложението-получател, ще видим, че то е получило правилно изпратения от приложението-изпращач обект от класа **Animal**:



Дълбоко копиране на обекти – пример

Настоящият пример илюстрира как можем да реализираме дълбоко копиране (deep copy) на обект, използвайки сериализация. Дълбокото копиране не само създава референция, но и клонира всички член-променливи на този обект и всички член-променливи на член-променливите на обекта и т.н. рекурсивно, за да нямат двата обекта нито една обща референция. По принцип създаването на дълбоко копие е нетривиален проблем, но решаването му чрез сериализация е лесно:

```
using System;
using System.IO;
```

```
using System.Text;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

[Serializable]
class SomeClass
{
   public StringBuilder mSomeStringBuilder;
   public string mSomeString;
   public object mSomeObject;
   public int mSomeInt;
   public SomeClass mSomeClass;
}

class DeepCopyDemo
{
   static void Main()
   {
      SomeClass original = new SomeClass();
      original.mSomeString = "Аз съм обикновено стрингче.";
      original.mSomeStringBuilder = new StringBuilder(
         "Защо този тип ме занимава с тия глупости?!");
      original.mSomeObject = new object();
      original.mSomeInt = 12345;
      original.mSomeClass = original;

      SomeClass copy =
         (SomeClass) DeepCopyDemo.DeepCopy(original);

      Console.WriteLine("copy.mSomeString={0}",
         copy.mSomeString );
      Console.WriteLine("copy.mSomeStringBuilder={0}",
         copy.mSomeStringBuilder);
      Console.WriteLine("copy.mSomeObject={0}",
         copy.mSomeObject);
      Console.WriteLine("copy.mSomeInt={0}\n", copy.mSomeInt );

      Console.WriteLine("copy.mSomeClass == copy ? {0}\n",
         Object.ReferenceEquals(copy.mSomeClass, copy) );

      Console.WriteLine("copy.mSomeClass == original ? {0}\n",
         Object.ReferenceEquals(copy.mSomeClass, original) );

      Console.WriteLine("Identical instances? {0}",
         Object.ReferenceEquals(copy, original));
      Console.WriteLine("Equal mSomeString? {0}",
         copy.mSomeString == original.mSomeString);
      Console.WriteLine("Equal mSomeString by reference? {0}",
         Object.ReferenceEquals(copy.mSomeString,
         original.mSomeString));
```

```
      Console.WriteLine("Equal mSomeStringBuilder? {0}",
        copy.mSomeStringBuilder == original.mSomeStringBuilder);
      Console.WriteLine(
        "Equal mSomeStringBuilder.ToString()? {0}",
        copy.mSomeStringBuilder.ToString() ==
        original.mSomeStringBuilder.ToString());
      Console.WriteLine("Equal mSomeObject? {0}",
        copy.mSomeObject == original.mSomeObject );
      Console.WriteLine("Equal mSomeInt? {0}",
        copy.mSomeInt == original.mSomeInt);
   }

   public static object DeepCopy(object aSourceObject)
   {
      IFormatter formatter = new BinaryFormatter();
      formatter.Context =
        new StreamingContext(StreamingContextStates.Clone);
      Stream memStream = new MemoryStream();
      formatter.Serialize(memStream, aSourceObject);
      memStream.Position = 0;
      object resultObject = formatter.Deserialize(memStream);
      return resultObject;
   }
}
```

Как работи примерът?

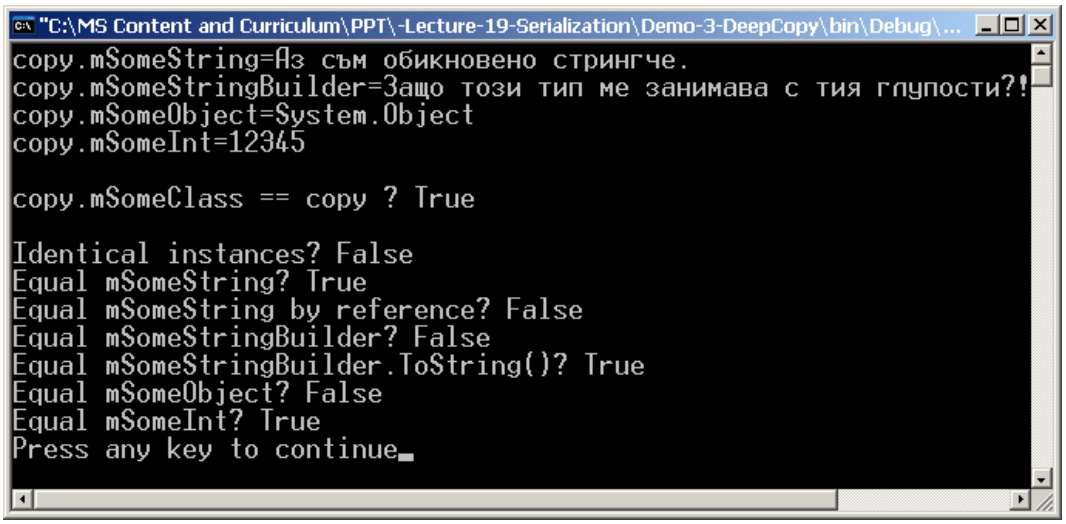
В началото на примера дефинираме класа **SomeClass**, който е сериализируем и съдържа няколко член-променливи от различни типове, включително и една член-променлива от собствения си тип **SomeClass** (имаме рекурсивно дефиниран клас). В примера ще направим дълбоко копие на обект от този клас.

В началото на функцията **Main()** създаваме обект от тип **SomeClass** и инициализираме член-променливите му със стойности. Забележете, че член-променливата **mSomeClass** съдържа референция към самия обект.

След инициализирането на член-променливите създаваме копие на обекта, като извикваме функцията **DeepCopy(…)** на класа. Тя създава дълбоко копие на подадения като параметър обект и връща това копие като резултат от извикването си. За да бъде създадено копието, обектът се сериализира в поток в паметта (**MemoryStream**) и след това се десериализира в нова инстанция. Член-променливите в десериализираното копие се създават правилно, понеже сериализиращият механизъм на CLR обхожда всички член-променливи и ги сериализира.

След като сме създали дълбоко копие, извеждаме съдържанието на член-променливите му и проверяваме доколко новополученият обект е точно копие на оригиналът. Резултатите от проверките също се извеждат в конзолата.

След изпълнение на примера, се получава следният резултат:



Резултатът показва, че оригиналът и копието, както и всички техни съставни части физически са разположени на различни места в паметта. Те нямат общи референции, т.е. реализирали сме дълбоко копиране на обекта.

IDeserializationCallback

Сериализацията се осъществява лесно, когато сериализираме обекти, които не зависят от други обекти. В реалността често обектите се сериализират заедно, като някои от тях зависят от другите. Това е проблем, понеже при десериализацията не е определен редът, в който се възстановяват обектите. В случаите, когато се налага да знаем кога е завършила десериализацията, за да извършим допълнителни действия върху десериализирания обект, можем да имплементираме интерфейса **`IDeserializationCallback`**.

Интерфейсът **`IDeserializationCallback`** съдържа един метод, който трябва да имплементираме – **`OnDeserialization(…)`**. CLR изпълнява този метод след пълната десериализация на обекта. В момента на изпълнение на метода е сигурно, че всички член-променливи са вече десериализирани.

IDeserializationCallback – пример

В настоящия пример ще бъде онагледено използването на интерфейса **`IDeserializationCallback`** за извършване на действия след десериализирането на даден обект:

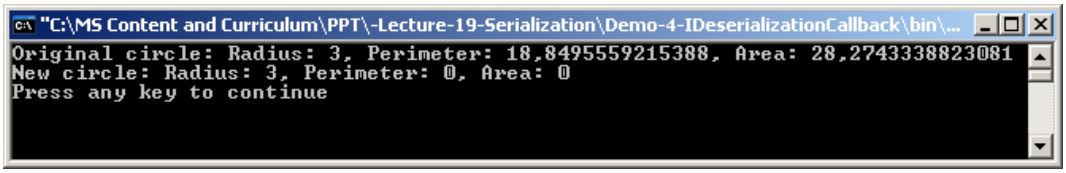
```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
```

```
namespace Demo_4_IDeserializationCallback
{
   [Serializable]
   class Circle //: IDeserializationCallback
   {
      private double mRadius;

      [NonSerialized]
      private double mPerimeter;

      [NonSerialized]
      private double mArea;

      public Circle(double aRadius)
      {
         mRadius = aRadius;
         InitInternalState();
      }

      private void InitInternalState()
      {
         mPerimeter = 2 * Math.PI * mRadius;
         mArea = Math.PI * mRadius * mRadius;
      }
/*
      void IDeserializationCallback.OnDeserialization(
         object aSender)
      {
         InitInternalState();
      }
*/
      public override string ToString()
      {
         string result= String.Format(
            "Radius: {0}, Perimeter: {1}, Area: {2}",
            mRadius, mPerimeter, mArea);
         return result;
      }
   }

   class IDeserializationCallbackDemo
   {
      static void Main()
      {
         Circle circle = new Circle(3.0);
         Console.WriteLine("Original circle: {0}", circle);

         IFormatter formatter = new BinaryFormatter();
         Stream stream = new MemoryStream();
```

```
        formatter.Serialize(stream, circle);
        stream.Position = 0;
        Circle newCircle =
           (Circle) formatter.Deserialize(stream);

        Console.WriteLine("New circle: {0}", newCircle);
     }
   }
}
```

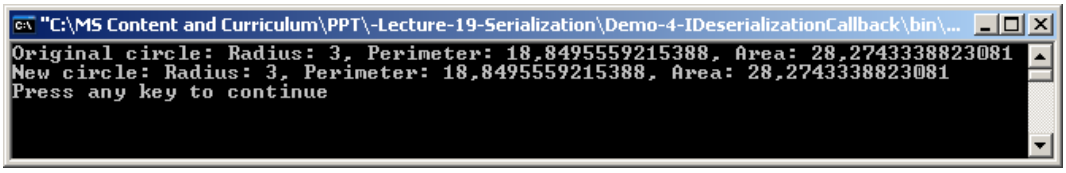
Проследяване на примера

Ако сега стартираме примера, ще получим следния резултат:



Трябва да обърнем внимание на това, че полетата за лице и параметър се губят, защото се сериализира и десериализира само радиусът.

Нека сега премахнем коментарите от заградения с тях код и изпълним отново примера. Този път десериализираният обект е коректно възстановен:



Как работи примерът?

Класът **`Circle`** описва геометричната фигура "кръг", която може да се сериализира като се съхрани само радиусът на кръга. Останалите полета са функции на този радиус и не е необходимо да се съхраняват, затова са маркирани с атрибута **`[NonSerialized]`**.

При десериализирането на обекта е необходимо всички характеристики (полета) на кръга да бъдат възстановени. Това ще бъде извършено от метода **`IDeserializationCallback.OnDeserialization(…)`**, който се извиква от CLR, след като обектът е създаден изцяло.

В примера се създава обект от тип **`Circle`** с определен радиус. Обектът се сериализира, след което се десериализира и съдържанието му се отпечатва в конзолата.

При първото изпълнение на примера, кодът свързан с имплементацията на интерфейса **`IDeserializationCallback`** е в коментари, поради което не се извиква функцията, възстановяваща полетата, които не се сериализират.

Това е причината полетата за лице и радиус да се губят при десериализацията.

След като премахнем коментарите около кода, свързан с имплементацията на интерфейса **IDeserializationCallback** и изпълним отново примера, виждаме, че полетата, които не са били сериализирани са възстановени коректно при десериализацията. След като сериализираните променливи са били възстановени и обектът е бил изцяло създаден, е изпълнен методът **IDeserializationCallback.OnDeserialization(…)**, с което са преизчислени лицето и параметъра на кръга.

ISerializable и контролиране на сериализацията

Има случаи, в които се налага да контролираме начина, по който се сериализират обектите. Например може да искаме да намалим обема на съхранената информация за обекта, особено, ако данните се записват във файл. За да предефинираме автоматичната сериализация, трябва да имплементираме интерфейса **ISerializable**, дефиниран в пространството **System.Runtime.Serialization**.

Имплементирайки интерфейса **ISerializable**, трябва да предоставим реализация на метода **GetObjectData(…)**, както и на специален конструктор, който ще бъде използван, когато обектът се десериализира. Те приемат едни и същи параметри – инстанция на класа **SerializationInfo** и инстанция на структура от тип **StreamingContext**.

Методът GetObjectData(SerializationInfo, StreamingContext)

При сериализацията на обект от клас, имплементиращ интерфейса **ISerializable**, форматерът извиква функцията **GetObjectData(…)**. Полетата, които ще бъдат сериализирани, се добавят в **SerializationInfo** обекта, подаден като параметър на функцията. Това става с помощта на метода **AddValue(…)** на този обект, който добавя полетата като двойки име/стойност. За име може да бъде използван произволен текст.

Ако нашият клас е наследен от базов клас, които имплементира интерфейса **ISerializable**, трябва да извикаме **base.GetObjectData(info, context)**, за да позволим на базовия обект да сериализира своите полета.

Конструкторът .ctor(SerializationInfo, StreamingContext)

По време на десериализацията чрез този специален конструктор на класа се подава **SerializationInfo** обект. За да възстановим състоянието на сериализирания обект, трябва да извлечем стойностите на полетата му от **SerializationInfo** обекта. Това става чрез имената, които сме използвали при сериализацията на полетата. Ако класът ни наследява клас, имплементиращ интерфейса **ISerializable**, трябва извикаме базовият конструктор, за да позволим на базовия обект да възстанови своите полета.

Не трябва да забравяме да имплементираме този конструктор, защото компилаторът няма как да ни задължи да го направим. Ако забравим да имплементираме конструктора, по време на десериализирането на обекта ще бъде хвърлено изключение.

Извличането на стойност от **SerializationInfo** обект става чрез подаването на името, асоциирано със стойността, на един от **GetXXX(…)** методите на **SerializationInfo**, където **XXX** се заменя с типа на стойността, която ще бъде извлечена - например **GetString(…)**, **GetDouble(…)** и др.

Контролиране на сериализацията – пример

Настоящият пример илюстрира нагледно, как можем да контролираме сериализацията, имплементирайки интерфейса **ISerializable**:

```
using System;
using System.Runtime.Serialization;

[Serializable]
class Person : ISerializable
{
   private string mName;
   private int mAge;

   private Person(SerializationInfo aInfo,
      StreamingContext aContext)
   {
      mName = (string)aInfo.GetString("Person's name");
      mAge = aInfo.GetInt32("Person's age");
   }

   void ISerializable.GetObjectData(SerializationInfo
      aInfo, StreamingContext aContext)
   {
      aInfo.AddValue("Person's name", mName);
      aInfo.AddValue("Person's age", mAge);
   }
}
```

Как работи примерът?

В примера дефинираме класа **Person**, който е сериализируем и съдържа две член-променливи – **mName** и **mAge**, чиито стойности ще запазим при сериализацията. Класът имплементира интерфейса **ISerializable**, което означава, че ще предостави собствена сериализация на полетата си.

Трябва да маркираме нашия клас с атрибута **[Serializable]**, въпреки че имплементираме интерфейса **ISerializable**. Без този атрибут CLR не счита, че инстанциите на класа могат да бъдат сериализирани.

Нашият клас имплементира интерфейса **ISerializable**, затова предоставяме реализация на метода **GetObjectData(…)** и на конструктора, който ще се извика при десериализацията.

В метода **GetObjectData(…)** добавяме стойностите на двете полета на класа в **SerializationInfo** обекта. Това става чрез метода **AddValue(…)**, на който подаваме името, което ще асоцииираме със стойността на променливата и самата променлива. Това име ще бъде използвано при десериализацията за извличане на стойността на променливата.

В конструктора на класа извличаме стойностите на променливите от **SerializationInfo** обекта. За целта използваме имената, които сме асоциирали със стойностите по време на сериализацията им. Прави впечатление, че в примера конструкторът за десериализация е деклариран като **private**, но това не е грешка, защото CLR може да извиква дори частни конструктори.

Конструкторът и методът **GetObjectData(…)** приемат като втори параметър **StreamingContext** обект, указващ къде се сериализира обектът. На **StreamingContext** структурата ще се спрем по-нататък в тази тема.

Ръчна сериализация с ISerializable – пример

Ще представим още един пример за ръчно сериализиране на обекти в .NET Framework чрез имплементация на интерфейса **ISerializable**:

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Soap;

namespace Demo_5_ISerializable
{
   [Serializable]
   public class Person : ISerializable
   {
      protected int mAge;
      protected string mName;

      public Person(string aName, int aAge)
      {
         mName = aName;
         mAge = aAge;
      }

      protected Person(SerializationInfo aInfo,
         StreamingContext aContext)
      {
         mName = aInfo.GetString("Person's name");
         mAge = aInfo.GetInt32("Person's age");
      }
```

```
    public virtual void GetObjectData(SerializationInfo aInfo,
      StreamingContext aContext)
    {
      aInfo.AddValue("Person's name", mName);
      aInfo.AddValue("Person's age", mAge);
    }
}

[Serializable]
sealed class Employee : Person
{
    private string mJobPosition;

    public Employee(string aName, int aAge,
      string aJobPosition) : base(aName, aAge)
    {
      mJobPosition = aJobPosition;
    }

    private Employee(SerializationInfo aInfo,
      StreamingContext aContext) : base(aInfo, aContext)
    {
      mJobPosition = aInfo.GetString("Employee's job");
    }

    public override void GetObjectData(SerializationInfo aInfo,
      StreamingContext aContext)
    {
      base.GetObjectData(aInfo, aContext);
      aInfo.AddValue("Employee's job", mJobPosition);
    }

    public override string ToString()
    {
      string value = String.Format(
        "(Name: {0}, Age: {1}, Job: {2})",
        mName, mAge, mJobPosition);
      return value;
    }
}

class ISerializableDemo
{
    static void Main()
    {
      Employee employee = new Employee("Jeffrey Richter",
        45, "CEO");
      Console.WriteLine("Employee = {0}", employee);
      FileStream empoyeeFile = new FileStream("employee.xml",
```

```
            FileMode.Create);
        using (empoyeeFile)
        {
            IFormatter formatter = new SoapFormatter();
            formatter.Serialize(empoyeeFile, employee);
            Console.WriteLine("Employee serialized.");

            empoyeeFile.Seek(0, SeekOrigin.Begin);
            Employee deserializedEmployee =
               (Employee) formatter.Deserialize(empoyeeFile);
            Console.WriteLine("Employee deserialized.");
            Console.WriteLine("Deserialized = {0}",
              deserializedEmployee);
        }
      }
   }
}
```

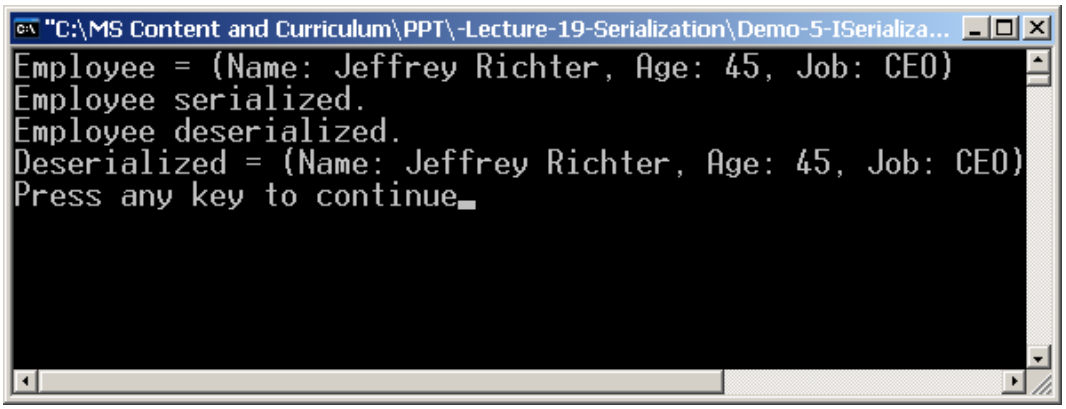
Как работи примерът?

В примера сме дефинирали клас **Person** и негов наследник – клас **Employee**. И двата класа имплементират интерфейса **ISerializable** и дефинират метод за сериализация **GetObjectData(SerializationInfo, StreamingContext)**, както и конструктор за десериализация със същата сигнатура.

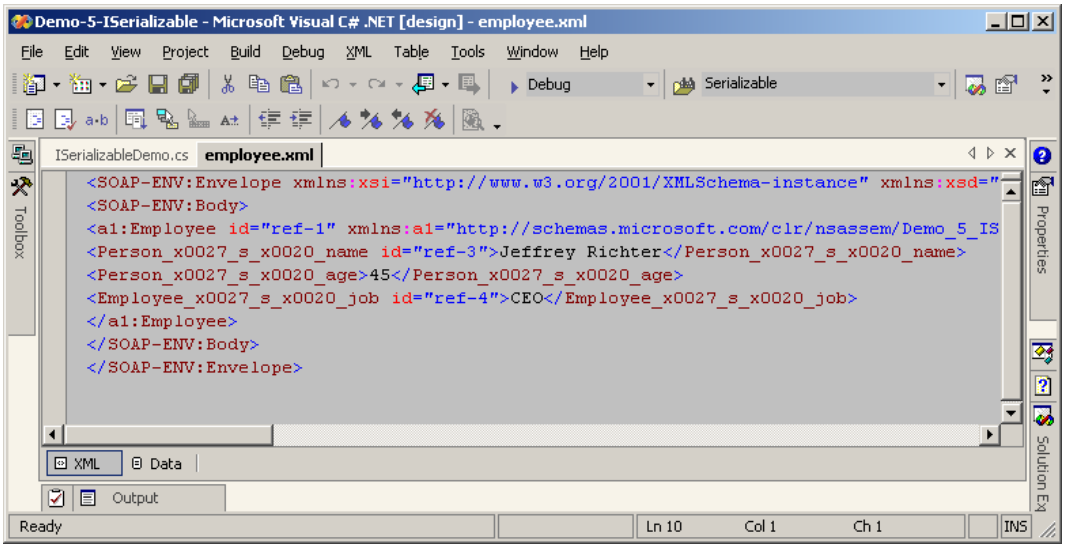
Класът **Person** е същият като в предишния пример, но сме добавили конструктор, който инициализира полетата му.

Класът **Employee** има една член-променлива **mJobPosition**. Първият конструктор служи за инициализация на полета на класа. В него той извиква конструктора на базовия клас и след това инициализира своето поле. Вторият конструктор се използва за десериализация на обекта, като за целта се извиква конструкторът за десериализация на базовия клас и след това се възстановява стойността на член-променливата **mJobPosition** от подадения **SerializationInfo** обект. В метода **GetObjectData(…)** първо се извиква **base.GetObjectData(…)**, за да може базовият клас да съхрани полетата си и след това се съхранява стойността на член-променливата **mJobPosition**. В класа е предефиниран метода **ToString()**, който връща символен низ, описващ съдържанието на обект от този тип.

За да демонстрираме работата на сериализацията и десериализацията, във функцията **Main()** на класа **ISerializableDemo** създаваме обект от класа **Employee** и отпечатваме съдържанието му в конзолата. След това създаваме **SoapFromatter**, с който сериализираме обекта в SOAP формат (ще го разгледаме в детайли в темата за уеб услуги и го записваме във файла **employee.xml**. Накрая десериализираме сериализирания обект и го отпечатваме в конзолата. Ето какъв е резултатът след изпълнението на примера:



Както виждаме, информацията е възстановена коректно и ръчно реализираните сериализация и десериализация работят успешно. Ето как изглежда и съдържанието на файла **`employee.xml`**, в който е записан сериализираният обект:



Имената на XML таговете се вземат от зададените при сериализацията имена, като символите, които не са допустими в имена на тагове се заменят със съответна escaping последователност.

Контекст на сериализация (Streaming Context)

Структурата **`StreamingContext`** се използва, за да се укаже къде се сериализира обектът. Тя има две публични свойства:

- **`Context`** – обект асоцииран с инстанция на **`StreamingContext`**. Тази стойност обикновено не се използва освен, ако не сме асоциирали интересна стойност с нея в процеса на сериализация.
- **`State`** – стойност от изброимия тип **`StreamingContextStates`**. По време на сериализацията това свойство указва къде се сериализира

обектът. Например, когато сериализираме във файл, стойността му ще бъде **File**. По време на десериализация, свойството указва от къде десериализираме данните.

Възможните стойности на **StreamingContextStates** и техните значения са следните:

- **CrossProcess** (0x0001) – данните се сериализират в друг процес на същия компютър.
- **CrossMachine** (0x0002) – данните се сериализират на друг компютър.
- **File** (0x0004) – данните се сериализират във файл.
- **Persistence** (0x0008) – данните се сериализират в база от данни, файл или друг носител.
- **Remoting** (0x0010) – данните се сериализират отдалечено на неопределено място, което може да е на друг компютър.
- **Other** (0x0020) – не е известно къде се сериализират данните.
- **Clone** (0x0040) – указва, че графът от обекти се клонира. Данните се сериализират в същия процес.
- **CrossAppDomain** – данните се сериализират в друг домейн на приложение.
- **All** (0x00FF) – сериализираните данни могат да са от всеки контекст.

Подавайки **StreamingContext** обект, форматерът дава информация как ще бъде използван сериализираният обект. Тази информация може да бъде използвана от обекта, за да определи как да сериализира данните си. В зависимост от това къде ще бъде сериализиран, обектът може да сериализира различен брой от полетата си, да направи допълнителна обработка на данните или примерно да хвърли изключение. Не всеки клас има нужда от такава допълнителна обработка, но форматерът ни предоставя необходимата информация и ако ни е нужна, може да я използваме.

За ефективността на сериализацията

Трябва да имаме предвид, че сериализацията е относително бавен процес, тъй като изследва типовете и извлича стойностите им чрез отражение (reflection). Ако трябва да извършваме четене и писане на огромен брой обекти и производителността е от важно значение, се препоръчва да се реализира ръчно записване на стойностите в поток и ръчно възстановяване на обектите вместо да се използва вградената в .NET сериализация. Примерен сценарий, в който е по-добре да се реализира ръчна сериализация е, когато разработваме приложение за мобилно устройство с ограничени ресурси (бавен процесор, малко памет и т.н.).

XML сериализация

До момента разгледахме класическата сериализация и десериализация на обекти. Нека сега се запознаем с още една възможност за съхраняване и възстановяване състоянието на обекти, която .NET Framwork предоставя на програмиста – XML сериализацията.

Какво е XML сериализация?

XML сериализация представлява записването на публичните полета на обект в XML формат с цел съхранение или пренасяне. Тя е част от вградената поддръжка на XML в .NET Framework. Обратният процес на XML сериализацията е XML десериализацията.

XML сериализацията създава някои ограничения, които трябва да имаме предвид. При нея се сериализират само публичните полета и не се запазва целостта на типа. XML сериализацията не може да се справи с циклично свързани графи от обекти. Могат да се сериализират всякакви обекти, но класът трябва да има конструктор без параметри.

Всъщност XML сериализацията не е сериализация в истинския смисъл на това понятие, защото не съхранява и възстановява пълното състояние на обектите, а само части от него.

XML сериализация – пример

В следващия пример ще илюстрираме как един клас може да сериализира данните си чрез XML сериализация:

```
public class Student
{
   public string mName;
   public int mAge;

   public void SerializeToXml(Stream aStream)
   {
      XmlSerializer xmlSerializer =
         new XmlSerializer(typeof(Student));
      xmlSerializer.Serialize(aStream, this);
   }

   public static Student DeserializeFromXml(Stream aStream)
   {
      XmlSerializer xmlSerializer =
         new XmlSerializer(typeof(Student));
      Student st = (Student) xmlSerializer.Deserialize(aStream);
      return st;
   }
}
```

Как работи примерът?

Класът **Student** има две публични полета – **mName** и **mAge**. Те трябва да са публични, за да могат да се запазят при XML сериализацията.

Реализирали сме метод **SerializeToXml(…)**, който сериализира данните на класа в XML формат в подадения му като параметър поток. За целта създаваме обект от класа **XmlSerializer** и извикваме метода му **Serialize(…)**, който сериализира инстанцията на класа в потока.

Методът **DeserializeFromXml(…)** служи за десериализиране на данните от подадения му като параметър поток. За целта създаваме обект от класа **XmlSerializer** и извикваме метода му **Deserialize(…)**, който десериализира данните от потока и връща десериализирания обект.

Проста XML сериализация – пример

Ще представим още един по-подробен пример, илюстриращ възможностите на .NET Framework за сериализация на обекти в XML формат чрез класа **XmlSerializer**:

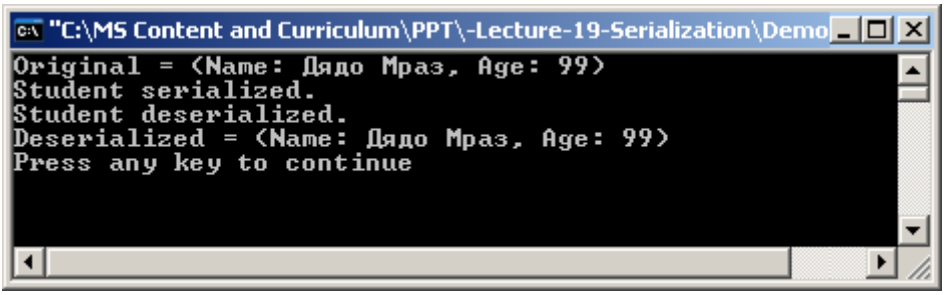
```
using System;
using System.IO;
using System.Xml.Serialization;

public class Student
{
   private string mName;
   private int mAge;

   public string Name
   {
      get { return mName; }
      set { mName = value; }
   }

   public int Age
   {
      get { return mAge; }
      set { mAge = value; }
   }

   public override string ToString()
   {
      string result =
         String.Format("(Name: {0}, Age: {1})", Name, Age);
      return result;
   }
}

class XmlSerializationDemo
```

```
{
   static void Main()
   {
      Student student = new Student();
      student.Name = "Дядо Мраз";
      student.Age = 99;
      Console.WriteLine("Original = {0}", student);

      // Serialize student object to "student.xml" file
      XmlSerializer xmlSerializer =
        new XmlSerializer(typeof(Student));
      FileStream outputStream = File.OpenWrite("student.xml");
      using (outputStream)
      {
        xmlSerializer.Serialize(outputStream, student);
      }
      Console.WriteLine("Student serialized.");

      // Deserialize student object from "student.xml" file
      FileStream inputStream = File.OpenRead("student.xml");
      using (inputStream)
      {
        Student deserializedStudent =
          (Student) xmlSerializer.Deserialize(inputStream);
        Console.WriteLine("Student deserialized.");
        Console.WriteLine("Deserialized = {0}",
          deserializedStudent);
      }
   }
}
```

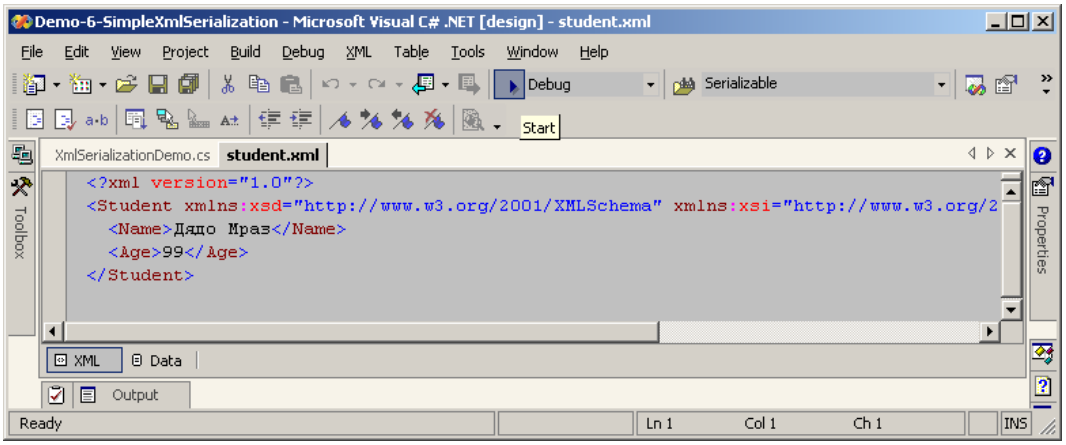
Как работи примерът?

В примера сме дефинирали класа **Student**, който има две публични свойства, които ще бъдат сериализирани. В класа е предефиниран методът **ToString()**, който връща символен низ, описващ съдържанието на обект от този тип. Този метод ще използваме за визуализация на **Student** обекти.

Във функцията **Main()** на класа **XmlSerializationDemo** създаваме обект от класа **Student**, инициализираме го и отпечатваме съдържанието му в конзолата. След това създаваме обект от класа **XmlSerializer** и използваме метода му **Serialize(…)**, за да сериализираме инстанцията на класа **Student** във файла **student.xml**. Накрая, използвайки метода **Deserialize(…)** на класа **XmlSerializer**, извършваме десериализацията от XML документ към инстанция на **Student** и отпечатваме съдържанието на тази инстанция в конзолата. Ето какъв е резултатът след изпълнението на примера:



Както виждаме, информацията е възстановена коректно. Оригиналният обект и обектът, получен след десериализацията, са еднакви. Ето как изглежда и съдържанието на файла **`student.xml`**, в който е записан сериализираният обект:



Виждаме, че в XML файла са записани всички публични членове на сериализирания **`Student`** обект.

Контролиране на изходния XML

Ако е нужно, можем да контролираме изходния XML, генериран от класа **`XmlSerializer`**. Това става чрез атрибути, които прилагаме към класа или към неговите полета. Ето кратък пример:

```
using System.Xml.Serialization;
public class OptionalOrder
{
   [XmlElement(ElementName = "Tax_Rate")]
   public decimal TaxRate;

   [XmlAttribute]
   public string FirstOrder;

   [XmlIgnoreAttribute]
   public bool FirstOrderSpecified;
```

```
   [XmlArrayAttribute("Items")]
   [XmlArrayItem("MemberName")]
   public OrderedItem[] OrderedItems;

   [XmlElement]
   public Employee[] Employees;
}
```

В примера сме дефинирали класа **OptionalOrder**. Към полетата му сме приложили атрибути, чрез които указваме как да се запишат в XML – чрез XML елементи, чрез XML атрибути и др.

Чрез атрибутът **XmlElement** указваме, че полето, към което е приложен, трябва да се сериализира като XML елемент. Чрез него можем да контролираме характеристиките на XML елемента, като най-често го използваме за указване на името на елемента.

Атрибутът **XmlAttribute** указва, че полето, към което е приложен, трябва да се сериализира като XML атрибут. По подразбиране **XmlSerializer** сериализира публичните полета като XML елементи.

Атрибутът **XmlIgnoreAttribute** указва, че полето не трябва да бъде сериализирано.

Атрибутът **XmlArrayAttribute** указва, че полето, към което е приложен, трябва да бъде сериализирано като масив. Чрез този атрибут може да укажем и името на генерирания XML елемент.

Атрибутът **XmlArrayItem** обикновено се използва заедно с атрибута **XmlArrayAttribute** и идентифицира тип, който може да се сериализира в масив. Чрез този атрибут също може да укажем името на генерирания XML елемент (както сме направили в нашия пример).

Контрол на XML сериализацията – пример

Ще представим още един, по-обширен, пример как чрез атрибути може да се контролира процесът на XML сериализацията:

```
using System;
using System.IO;

using System.Runtime.Serialization;
using System.Xml.Serialization;

[XmlRoot("animal")]
public class Animal
{
   [XmlArray("eyes")]
   [XmlArrayItem("eye")]
```

```
   public Eye[] Eyes;
   [XmlElement("claws")]
   public Claw[] Claws;
   [XmlIgnore]
   public string SomeMember = "Some member";

   public Animal Friend;
}

public class Eye
{
   [XmlAttribute("vision")]
   public double Vision;

   public Eye()
   {
   }

   public Eye(double aVision)
   {
      Vision = aVision;
   }
}

public class Claw
{
   [XmlElement(ElementName="claw")]
   public string Description;

   public Claw()
   {
   }

   public Claw(string aDescription)
   {
      Description = aDescription;
   }
}

public class ControllingSerializationDemo
{
   public static void SerializeAnimalToXml(Animal aAnimal,
      string aFileName)
   {
      XmlSerializer xmlSerializer =
         new XmlSerializer(typeof(Animal));
      TextWriter writer = new StreamWriter(aFileName);
      using (writer)
      {
         xmlSerializer.Serialize(writer, aAnimal);
```

```
    }
  }

  public static Animal DeserializeAnimalFromXml(
    string aFileName)
  {
    TextReader reader = new StreamReader(aFileName);
    using (reader)
    {
      XmlSerializer xmlSer = new XmlSerializer(typeof(Animal));
      object deserializedAnimal = xmlSer.Deserialize(reader);
      return (Animal) deserializedAnimal;
    }
  }

  public static void Main()
  {
    Animal animal1 = new Animal();
    animal1.Eyes = new Eye[] {new Eye(1.05), new Eye(0.85)};
    animal1.Claws = new Claw[] {
      new Claw("Left claw"),
      new Claw("Right claw")};

    Animal animal2 = new Animal();
    animal2.Eyes = new Eye[] {new Eye(1.00), new Eye(1.00)};
    animal2.Claws = new Claw[] {new Claw("Beautiful claw")};

    animal1.Friend = animal2;
    // animal2.Friend = animal1;

    SerializeAnimalToXml(animal1, "animal.xml");
    Console.WriteLine("Animal serialized.");

    Animal deserializedAnimal =
      DeserializeAnimalFromXml("animal.xml");
    Console.WriteLine("Animal deserialized.");
  }
}
```

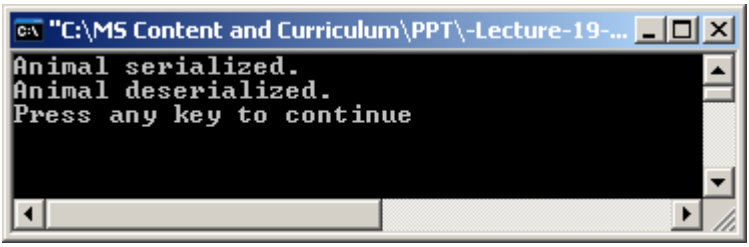
Как работи примерът?

Класът **Animal** съдържа няколко полета, за които сме указали чрез атрибутите **XmlArray**, **XmlArrayItem**, **XmlElement** и **XmlIgnore** как трябва да се запишат в изходния XML.

Класовете **Eye** и **Claw**, които се използват от класа **Animal** също ползват атрибути, за да опишат как да се запишат в изходния XML.

В класа **ControllingSerializationDemo** са реализирани два метода – **SerializeAnimalToXml** и **DeserializeAnimalFromXml**, които съответно сериализират и десериализират **Animal** обекти.

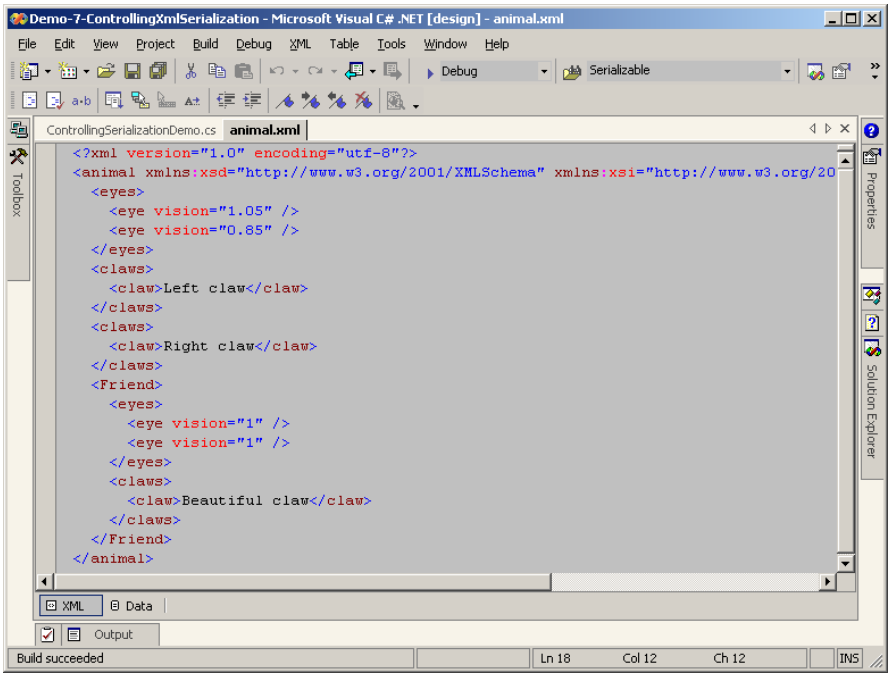
Във метода **Main()** създаваме две инстанции на класа **Animal**, задаваме стойности на публичните им членове и правим едната инстанция член на другата. След това извършваме сериализация във файла **animal.xml** и десериализираме този файл, за да получим обратно записаната в него **Animal** инстанция. След като изпълним примера получаваме следният резултат:



Обектът бива сериализиран и след това обратно десериализиран. На картинката по-долу виждаме как изглежда и файлът **animal.xml**, получен при сериализацията на обекта.

Забелязва се, че полето **SomeMember** не е било сериализирано, понеже е маркирано с атрибута **XmlIgnore**. Имената на елементите са такива, каквито сме указали чрез атрибутите, които сме приложили към полетата.

Ако в горния пример премахнем коментара от реда "**animal2.Friend = animal1**" и така направим двете инстанции на класа **Animal** циклично свързани една с друга и изпълним след това примера, ще получим изключение. Това се случва, защото XML сериализацията не може да сериализира циклични структури.



Външен контрол на XML сериализацията

В .NET Framework е предвиден механизъм, който ни позволява да контролираме XML сериализацията извън обекта, т.е. без да указваме това в изходния код на класа. Този механизъм се използва, когато нямаме достъп до изходния код на класа или когато искаме да създадем един набор от сериализируеми класове, но да сериализираме обектите по различен начин в зависимост от това къде се използват.

Външният контрол на сериализацията прилича много на контрола на сериализацията с атрибути. Функционалността е същата като при нея, дори класовете са същите, само механизмът за добавяне е различен.

Външният контрол на сериализацията се извършва чрез класовете **XmlAttributesOverrides** и **XmlAttributes**. Чрез тях, за всеки член на даден клас, се задава колекция **XmlAttributes**, описваща формата на изходния XML. За целта се създава **XmlAttributesOverrides** обект, който по-късно се подава на конструктора на **XmlSerializer**. Резултатният **XmlSerializer** обект използва информацията, която се съдържаща в **XmlAttributesOverrides**, за да определи как да извърши сериализацията. **XmlAttributesOverrides** обекта съдържа колекция от типове, за които ще бъде предефинирана автоматичната сериализация, както и **XmlAttributes** обект, асоцииран с всеки един от тях. **XmlAttributes** обектът съдържа избран набор от атрибути, указващи как да бъдат сериализирани всяко едно поле, свойство или клас.

Нека разгледаме следващия фрагмент код, илюстриращ как става това:

```
XmlAttributeOverrides overrides = new XmlAttributeOverrides();
XmlAttributes attribs = new XmlAttributes();
attribs.XmlElements.Add(new XmlElementAttribute("PersonName"));
overrides.Add(typeof(Person), "Name", attribs);
XmlSerializer xmlSerializer =
  new XmlSerializer(typeof(Person), overrides);
...
```

В примера указваме на XML сериализацията, че полето (или свойството) **Name** на класа **Person** трябва да се запише в XML таг с име **PersonName**.

Първо създаваме **XmlAttributesOverrides** обект. След това създаваме **XmlAttributes** обект и към колекцията му **XmlElements** добавяме нов **XmlElementAttribute**. После, използвайки метода **Add(…)**, добавяме **XmlAttributes** обекта към **XmlAttributesOverrides** обекта. Като параметри на метода подаваме и типа, за който предефинираме сериализацията, както и името на полето, чиято сериализация предефинираме. Накрая подаваме **XmlAttributesOverrides** обекта на конструктора на **XmlSerializer**.

Външен контрол на сериализацията – пример

Ще представим един пример, илюстриращ, как можем да контролираме формата на изходния XML документ при XML сериализация по недекларативен път (без да се променя сорс кода на класа, който се сериализира):

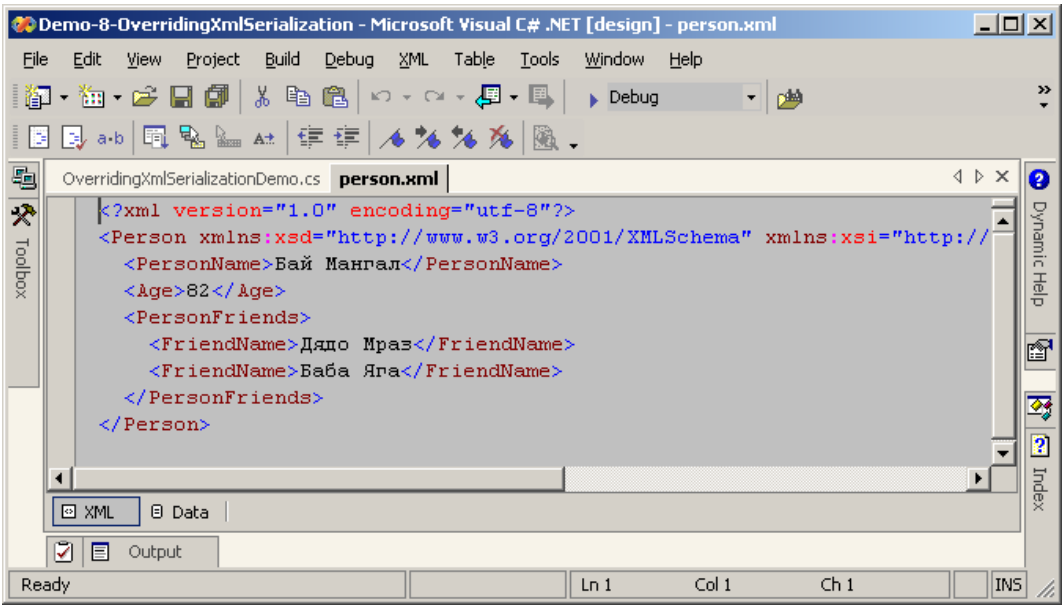
```
using System;
using System.IO;
using System.Xml.Serialization;

public class Person
{
   public string Name;
   public int Age;
   public string[] Friends;
}

class OverridingXmlSerializationDemo
{
   static void Main()
   {
      Person person = new Person();
      person.Name = "Бай Мангал";
      person.Age = 82;
      person.Friends = new string[] {"Дядо Мраз", "Баба Яга"};

      XmlAttributeOverrides overrides =
         new XmlAttributeOverrides();

      XmlAttributes nameAttributes = new XmlAttributes();
      XmlElementAttribute nameElement =
         new XmlElementAttribute("PersonName");
      nameAttributes.XmlElements.Add(nameElement);
      overrides.Add(typeof(Person), "Name", nameAttributes);

      XmlAttributes friendsAttributes = new XmlAttributes();
      XmlArrayAttribute friendsArray =
         new XmlArrayAttribute("PersonFriends");
      friendsAttributes.XmlArray = friendsArray;
      XmlArrayItemAttribute friendsArrayItem =
         new XmlArrayItemAttribute();
      friendsArrayItem.ElementName = "FriendName";
      friendsAttributes.XmlArrayItems.Add(friendsArrayItem);
      overrides.Add(typeof(Person), "Friends",
         friendsAttributes);

      TextWriter writer = new StreamWriter("person.xml");
      using (writer)
      {
         XmlSerializer xmlSer = new XmlSerializer(typeof(Person),
            overrides);
```

```
            xmlSer.Serialize(writer, person);
        }
        Console.WriteLine("Person instance serialized.");
    }
}
```

Как работи примерът?

Дефинирали сме клас **Person** с няколко полета. В началото на функцията **Main()** създаваме инстанция на класа **Person** и инициализираме нейните полетата. След това на полето **Name** от класа **Person** съпоставяме колекция от XML атрибути, които указват, че това поле трябва да се форматира като XML елемент с име **PersonName**. После на полето **Friends** от класа **Person** (което представлява масив от низове) съпоставяме колекция от XML атрибути, които указват, че това поле трябва да се форматира като XML елемент с име **PersonFriends**, което съдържа в себе си за всеки елемент от масива по един XML елемент с име **FriendName**. Накрая сериализираме обекта във файла **person.xml**.

Ето как изглежда и файлът **person.xml**, получен при сериализацията:



Виждаме, че полетата са сериализирани по начина, който сме указали чрез атрибутите, които сме приложили към тях.

Приложение: FormatterServices

Ще разгледаме съвсем накратко, без да даваме пример, средствата за реализация на собствени форматери в .NET Framework. Едно от тези средства е класът **FormatterServices**. Той предоставя основната функ-

ционалност, която трябва да притежава форматера – извличане на сериализируемите членове на обект, определяне на техните типове и извличане на стойностите им. Този клас не може да бъде наследяван.

Методи за сериализация

public static MemberInfo[] GetSerializableMembers(Type)

Методът приема като параметър типа на класа, който ще бъде сериализиран, и връща като резултат масив от **`MemberInfo`** обекти, съдържащи информация за сериализируемите членове на класа.

public static Object[] GetObjectData(Object, MemberInfo[])

Методът приема като параметри обект, който ще бъде сериализиран и масив с членовете, които трябва бъдат извлечени от обекта. За всеки от тях се извлича стойността, асоциирана с него в сериализирания обект и тези стойности се връщат като масив от обекти. Дължината му е същата, като дължината на масива с членовете, извличани от обекта.

Методи за десериализация

public static Type GetTypeFromAssembly(Assembly, String)

Методът намира типа на определен обект в дадено асембли. Той приема като параметри асемблито и името на обекта, който ще се търси, и връща като резултат типа на този обект.

public static Object GetUninitializedObject(Type)

Методът приема като параметър тип на обект и връща като резултат нова инстанция на обект от дадения тип.

public static Object GetObjectMembers(Object, MemberInfo[], Object[])

Методът попълва със стойности полетата на обект, като тези стойности се вземат от масив с обекти. За целта като параметри му се подават обекта, чиито полета ще се запълват, масив от **`MemberInfo`** обекти, описващ кои полета да се запълват и масив с обекти, от който ще се вземат стойностите за полета. Като резултат се връща обекта с попълнени полета.

Упражнения

1. Да се дефинира клас **`Graph`**, който описва насочен граф (представен като масив от върхове). Да се дефинира клас **`Node`**, който описва един връх от графа. Класът **`Node`** трябва да съдържа информационна част (текстово поле) и масив от наследници (инстанции на същия клас **`Node`**). Да се Реализира функционалност, която сериализира и десериализира инстанции на класа **`Graph`**.

2. Опитайте се да сериализирате бинарно инстанция на класа **`System.Collections.Hashtable`**. Опитайте след това да сериализирате хеш-таблица с XML сериализация. Какви проблеми възникват? Можете ли да обясните защо XML сериализацията не работи? Предложете алтернативно решение.
3. Дефинирайте класове **`Country`** и **`Town`**, които съдържат информация за държави и градове. Може да считате, че в една държава има много градове. Реализирайте бинарна и XML сериализация и десериализация за тези класове. Реализирайте TCP сървър, който по име на държава връща информация за държавата заедно с всички градове в нея (във вид на бинарно сериализиран **`Country`** обект). Реализирайте Windows Forms клиентско приложение за TCP сървъра, което позволява да се извлича и визуализира информация за държавите. Клиентът и сървърът трябва да поддържат два режима на работа – с бинарна сериализация и с XML сериализация.
4. Обяснете защо **`SoapFormatter`** може да сериализира цикличен граф от обекти, а XML сериализацията не може. Упътване: създайте цикличен граф от обекти, сериализайте го по двата начина и сравнете изходните XML файлове.

Използвана литература

1. Михаил Стойнов, Сериализация на данни – http://www.nakov.com/dotnet/lectures/Lecture-19-Serialization-v1.0.ppt
2. MSDN Library – http://msdn.microsoft.com
   - Object Serialization in the .NET Framework
   - System.Runtime.Serialization Namespace
   - System.Runtime.Serialization.Formatters Namespace
   - System.Xml.Serialization Namespace
   - XML and SOAP Serialization
   - XmlSerializer Class
   - Controlling XML Serialization Using Attributes
   - Attributes That Control Encoded SOAP Serialization
   - Attributes That Control XML Serialization
   - The XML Schema Definition Tool and XML Serialization
   - Generating SOAP Messages With XML Serialization
   - FormatterServices Class
3. Vyacheslav Biktagirov, .NET Serialization – http://www.csharphelp.com/archives/archive38.html
4. Mickey Williams, CodeGuru: .NET Serialization - http://www.codeguru.com/columns/DotNet/article.php/c6595/

Глава 21. Уеб услуги с ASP.NET

Автори

Деян Варчев

Стефан Добрев

Необходими знания

- Базови познания за .NET Framework
- Базови познания за езика C#
- Базови познания за ASP.NET
- Начални умения за работа с Visual Studio .NET
- Познания по XML
- Атрибути

Съдържание

- **Инфраструктурата на уеб услугите**
  - Разпределени приложения
  - Нуждата от уеб услуги
  - Услуги и уеб услуги
  - UDDI директории за уеб услуги
  - Откриване на уеб услуги (DISCO)
  - WSDL описания на услуги
  - SOAP – формат на заявките
  - Протоколен стек на уеб услугите
  - Сценарии за използване на уеб услуги
  - .NET Enterprise приложения
- **Уеб услугите в ASP.NET**
  - Архитектура
  - Създаване и публикуване на уеб услуги
  - Използване на уеб услуги. Генериране на междинен (прокси) клас
  - Уеб услугите и VS.NET – създаване и консумиране
  - Атрибути за уеб услугите – **`[WebService]`**, **`[WebMethod]`**
  - Прехвърляне на типове (type marshalling)

- Разгръщане (deployment) на уеб услуги върху IIS
- Дебъгване на уеб услуги
- Моделът на изпълнение на уеб услугите в ASP.NET
- Асинхронно извикване
- Уеб услуги и работа с данни
- Поддръжка на сесии
- Сигурност на уеб услугите. Сигурност чрез сесии
- Изключенията в уеб услугите

В тази тема ...

В настоящата тема ще разгледаме уеб услугите и работата с тях чрез средствата на .NET Framework и ASP.NET. Ще изясним концепциите и стандартите, които стоят в основата на уеб услугите, и ще обясним защо те са се превърнали в стандарт за интеграция и междуплатформена комуникация. Ще се запознаем с различни сценарии за използването им. Ще разгледаме приложението на уеб услугите за изграждане на многослойни .NET Enterprise приложения. Ще разгледаме програмния модел за уеб услуги в ASP.NET и средствата за тяхното изграждане, изпълнение и разгръщане (deployment). Ще се спрем и на някои често срещани проблеми и утвърдени практики при разработката на уеб услуги чрез .NET Framework и ASP.NET.

Възникването на уеб услугите

В зората на Интернет основна цел е била да се направят публично достъпни определени документи, статии и други ресурси за хора, които са били заинтересовани от тяхното съдържание. С бързото развитие на Интернет технологиите в края на 90-те години Интернет става място не само за уеб страници, но и единно място за обмяна на съобщения и информация между различни приложения. Липсата на единен стандарт за описанието и разпространението им, както и нуждата от адаптери за интеграция на вече съществуващите технологии, пораждат изграждането на нов независим (както от самото приложение така и от платформата, на която е разположен) стандарт – SOAP (Simple Object Access Protocol). Днес работата на всяко уеб базирано приложение, което е отворено към света, е немислима без уеб услугите, защото те са се превърнали в стандарт за междуплатформена комуникация и интеграция и се основават на вече утвърдили се в глобалната мрежа модели и стандарти.

Разпределени приложения

В днешно време повечето приложения се състоят от няколко отделни компонента, които взаимодействат помежду си, и заедно решават една обща задача. Чрез разделянето на няколко съставни части, логиката на самото приложение се разпределя между отделните му компоненти, всеки от които е логически обособен, има ясна отговорност и може да е разположен физически на отделен компютър. Оттук идва и името на самите приложения – разпределени. Основен принцип при съставянето на всеки компонент е той да изпълнява добре дефинирана задача (strong cohesion) и да е логически независим (loosely coupled) от останалите компоненти.

Модели за разпределени приложения

С годините еволюцията на софтуерните технологии е преминала през различни модели на разпределени приложения, всеки от които има своите силни и слаби страни. Да разгледаме някой от тях:

- **Модел "Клиент/Сървър"** – при този модел приложението е двуслойно. На сървъра са разположени данните за системата и общата за всички логика, а при клиента стои приложение, което взаимодейства с потребителите и комуникира със сървъра. Типичен случай на такава система е сървър с база от данни и множество клиенти, които работят с общите данни от сървъра.
- **Модел "Разпределени обекти"** – този модел предоставя възможност за отдалечен достъп до обекти, като позволява създаване на обекти върху отдалечен сървър и извикване техни методи. Ето някои архитектури, които използват този модел:
  - **DCOM** (Distributed Component Object Model) – представлява разширение на COM модела в Windows операционни системи, което

позволява COM компоненти, инсталирани на отдалечени една от друга машини, да комуникират помежду си. COM/DCOM архитектурата е разработена от Microsoft и въпреки, че е пренесена и върху други платформи, нейното основно предназначение си остава най-вече за операционните системи на Microsoft Windows.

- **CORBA** (Common Object Request Broker Architecture) – представлява отворен стандарт за комуникация между обекти, разположени върху отдалечени една от друга машини. Стандартът е разработен от консорциума OMG (Object Management Group). Въпреки, че прави комуникацията независима както от езика, на който са написани приложенията, така и от операционната система, върху която се изпълняват, CORBA не е набрал популярност заради голямата си сложност и трудността за имплементация.
- **Java RMI** (Remote Method Invocation) – представлява стандарт за разпределени приложения, разработен от Sun, и базиран на Java платформата. Позволява комуникация между отдалечени обекти, разработени на Java, чрез отдалечено извикване на методите им. За разлика от CORBA и DCOM, RMI е значително по-опростен, но работи само с Java обекти.
- **.NET Remoting** – представлява технология, използвана в .NET Framework, която осигурява лесен и прозрачен достъп до отдалечени .NET обекти. Работи само с .NET обекти.

- **Модел "Уеб услуги"** – базиран е изцяло на отворени стандарти за отдалечени извиквания, в чиято основа стои XML. Най-често за комуникацията се използват HTTP протоколът и моделът заявка-отговор, което прави Интернет и WWW идеални за преносна среда на уеб услугите, а от там идва и името им. Уеб услугите се самоописват чрез езика WSDL и това значително опростява използването им.

Уеб услугите са настоящето и бъдещето на разпределените приложения. В самата си същност те представляват функционално независими програмни компоненти и извеждат междуплатформената комуникация на ново ниво на абстракция, което е зависимо от компанията-производител, използвания програмен език или софтуерна платформа.

Нуждата от уеб услуги

Вече разгледахме някои от вече съществуващите модели за разпределени приложения и изтъкнахме част от недостатъците им. Сега ще се спрем по-подробно на нуждата от уеб услуги и ще изясним защо се е стигнало до тяхното създаване.

Недостатъци на модела "Клиент/сървър"

Моделът клиент-сървър (двуслойна архитектура) не пасва добре на идеята за разпределените приложения, защото с нарастване на сложността им нараства и нуждата от създаването на повече от два слоя.

Остава възможността да се използва модел за отдалечена комуникация, които позволява изграждането на многослойни разпределени приложения. Двата най-често използвани подхода за това са "Разпределени обекти" и "Уеб услуги".

Недостатъци на модела "Разпределени обекти"

С широкото навлизане на Интернет и неговото масово използване се е зародила нуждата от разпределени приложения, които да комуникират помежду си посредством глобалната мрежа.

Моделът "Разпределени обекти" не е създаден с презумпцията, че трябва да използва Интернет като преносна среда. Всеки един от разгледаните разновидности на модела е разчитал на свой собствен протокол за пренасяне на информацията. Добавяйки наличието на защитни стени (firewalls) в Интернет пространството, комуникацията между отделните приложения става силно затруднена.

Основен проблем при технологиите тип "Разпределени обекти" са липсата на междуплатформена съвместимост (interoperability) и трудностите при изграждането на хетерогенна инфраструктура за предоставената услуга. Използването на отдалечен обект или негов метод изисква, при клиента да е имплементирана същата архитектура, каквато и на сървъра, а това води до силна технологична обвързаност между доставчика на услугата и нейните консуматори.

Още един проблем на разглеждания модел е поддръжката на различни версии и настройки на приложението. За да може клиентът да използва даден отдалечен обект, той трябва да е съобразен с версия на приложението, което е разположено на отдалечената машина, както да използва и идентични настройки с него.

Изисквания за съвременните разпределени приложения

Недостатъците на модела "Разпределени обекти" формират изисквания, на които трябва да отговаря съвременната архитектура за разпределени приложения. Някои от тях са следните:

- Междуплатформена комуникация – отдалечените програмни компоненти трябва да са достъпни за клиенти с различни операционни системи, изградени върху различни софтуерни платформи и с различни езици за програмиране.
- Базирана на отворени Интернет стандарти и технологии – различните компоненти на разпределените приложения трябва да са лесно достъпни през Интернет и да се възползват изцяло от предимствата на глобалната мрежа. Те трябва да не са технологично обвързани с даден доставчик.
- Самоописание – архитектурата за разпределени приложения трябва да предоставя възможност за самоописание на програмните компоненти, което да позволява тяхното използване без да е необходимо

предварително познаване на структурата им и интерфейсът за достъп до тях.

Уеб услугите решават всички тези проблеми, а освен това откриват и нови хоризонти пред разработчиците на разпределени приложения. Нека ги разгледаме в детайли.

Уеб услуги

Уеб услугите са нова ера в разработката на разпределени приложения. Те предоставят ново ниво на абстракция над вече съществуващите модели, което стои над езиците за програмиране, операционните системи и мрежовите комуникационни протоколи. Възползвайки се от вече изградените технологични модели в Интернет и базирайки се изцяло на отворени стандарти, уеб услугите се превръщат в основната инфраструктура, която свързва всички компютърни устройства.

Преди да се спрем по-подробно на технологията на уеб услугите, нека първо обясним какво всъщност означава терминът "услуга".

Какво е услуга?

В реалния живот услугата представлява единица работа извършена от доставчика на услуги. На всеки от нас му се случвало да му се развали телевизора или да му се запуши водопроводен канал. В такъв случай ние извикваме техник и той трябва да реши проблема, като представи пред нас желания резултат – поправен телевизор или отпушен канал.

В описаните сценарии ние се явяваме клиенти на услугата, т.е. нейни консуматори, а фирмата, за която работи техникът, неин доставчик. Услугата има ясно дефинирани входни параметри и ясна цел (изходни резултати). Тя има различни качествени характеристики: цена на самата услуга, време за нейното извършване, коректност при изпълнението ѝ и други. Услугата е лесна за използване – ние не се интересуваме по какъв начин нашият телевизор ще бъде поправен или колко усилия ще изразходва водопроводчикът за да отпуши канала – за нас е важно работата да бъде свършена. Услугата е и винаги достъпна при нужда от нея.

Какво е уеб услуга?

Уеб услугите не само наподобяват услугите от реалния живот, но и моделират тяхното поведение. Те представляват програмни компоненти (някаква специфична логика, изчислителен ресурс или определена информация), които са достъпни отдалечено през уеб.

Уеб услугите са достъпни на практика от всеки клиент, който поддържа връзка с уеб, защото използват отворени Интернет стандарти за комуникация. Те са независими както от операционната система, така и от платформата и езиците за програмиране, на които се разработват.

Архитектурно уеб услугите са функционално независими компоненти и са слабо обвързани с клиента, който ги използва (loosely coupled). Клиентът поръчва, услугата изпълнява поръчката и връща резултата обратно при клиента. Клиентът не се интересува как точно работи уеб услугата и за да я използва не трябва да знае нищо повече за нея освен какви входни данни да й подаде.

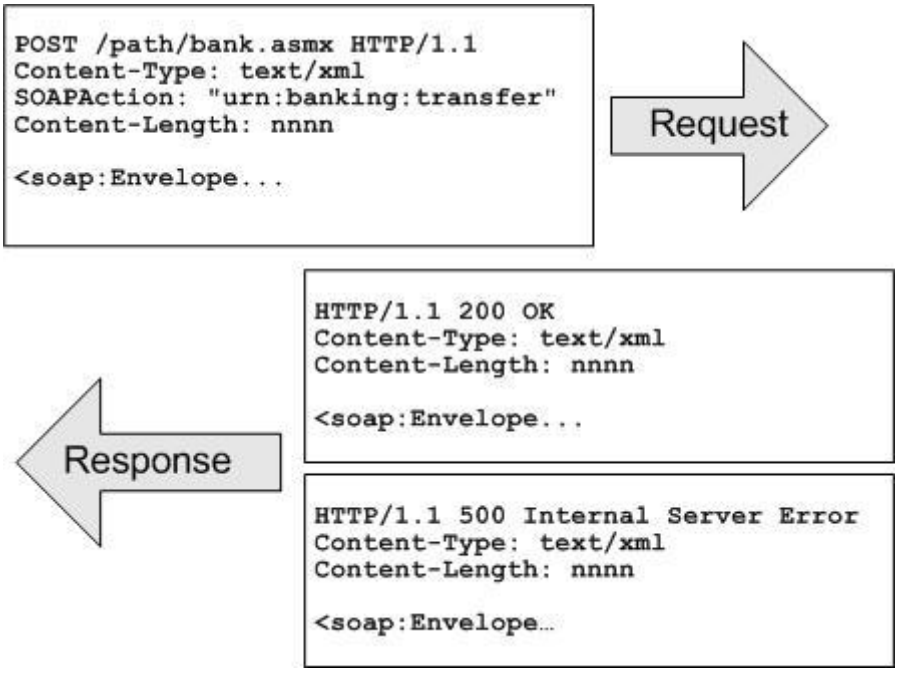
Принцип на действие на уеб услугите

Уеб услугите представляват XML базиран стандарт за отдалечено извикване на функционалност. Те работят на принципа на обмяна на прости SOAP съобщения между клиента и доставчика на услугата. Всяко съобщение се състои от данни и метаданни, описващи тези данни. Ще се спрем по-подробно на стандарта SOAP и на структурата на SOAP съобщенията малко по-нататък, когато разглеждаме инфраструктурата на уеб услугите.

Уеб услугите използват утвърдения в Интернет и при уеб технологиите модел "заявка/отговор" (request/response), т. е. за всяка една отделна заявка към сървъра, той връща отделен отговор специално за нея. По същия модел работят и уеб приложенията: уеб клиентът подава HTTP заявки, а уеб сървърът ги обработва и връща HTTP отговор.

Пренос на SOAP съобщения по HTTP

При уеб услугите протоколът за пренос на заявките и отговорите по подразбиране е HTTP, но като такъв може да се използва и всеки друг протокол, който може да пренася XML данни. Следващата фигура илюстрира използването на HTTP за пренос на SOAP съобщения:



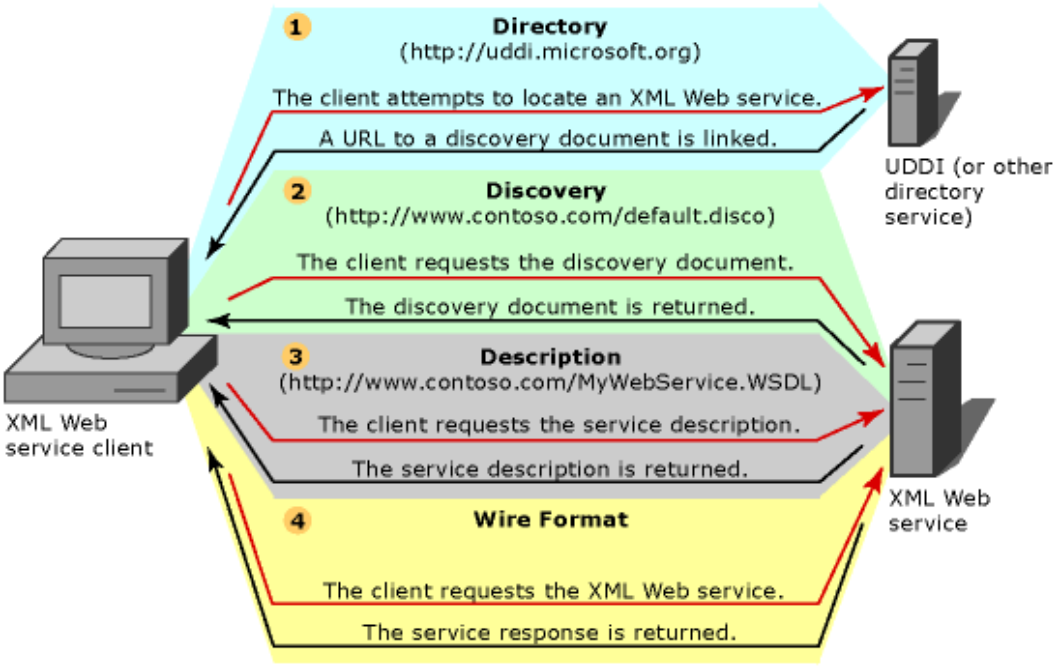
HTTP заявката се състои от две части: хедър, който съдържа различни параметри на заявката (информация за самата заявка и за клиента, който я изпраща) и тяло, което съдържа SOAP съобщението. SOAP съобщението се състои също от две части: данни (SOAP body) и метаданни (SOAP header).

В хедъра на HTTP заявката се посочва нейният вид. В примера е използвана HTTP-POST заявка по версия 1.1 на HTTP протокола. В хедъра се задава още типът на съдържанието (**`Content-Type`**), който трябва да е **`text/xml`**, тъй като SOAP съобщенията представляват XML. Заявката задължително трябва да съдържа и хедъра **`SOAPAction`**, дори и ако той е без съдържание. Неговото предназначение е да укаже същността на SOAP съобщението.

След като получи така формираната заявка, сървърът изпраща отговор, който може да е или със статус **`200 OK`** (успех), или статус **`500 Internal Server Error`** (грешка). Грешка се връща, ако SOAP съобщението, изпратено като отговор, съдържа SOAP Fault, т. е. възникнал е проблем (изключение) при изпълнението на услугата.

Инфраструктура на уеб услугите

След като проследихме в детайли как се транспортират SOAP заявките и съответните им отговори, сега ще разгледаме цялата инфраструктура на уеб услугите – съвкупността от стандартите, моделите и принципите, на които те се базират.



Източник: MSDN Library

Инфраструктурата на уеб услугите е изградена върху няколко основни принципа: тя прави услугите лесно достъпни, самоописващи се и използващи вече утвърдени и стандартизирани протоколи за комуникация.

На схемата по-горе са представени отделните компоненти от цялостната инфраструктура на уеб услугите:

1. Директория (Directory) – представлява централизирано място (каталог) за съхранение на описания на уеб услуги, разработени от различни производители. Предоставя възможност за търсене на услуги по различни параметри. Използва се стандартът UDDI (Universal Description, Discovery, and Integration), който служи за регистрация, откриване и свързване към конкретна уеб услуга.
2. Откриване (Discovery) – това е процесът на намиране описанието на дадена уеб услуга. DISCO спецификацията предоставя начин за откриване на описанията на уеб услуги, разположени на определен сървър.
3. Описание (Description) – за да можем да използваме определена уеб услуга трябва да знаем нейното описание (програмен интерфейс). Описанието се изготвя по стандартизиран начин чрез използване на XML базирания език за описание на интерфейса на уеб услуги – WSDL (Web Services Description Language).
4. Формат на заявките (Wire Format) – за да бъдат универсални уеб услугите се нуждаят от стандарти и протоколи, които са утвърдили мястото си в Интернет пространството. Това са XML, XSD, HTTP и SOAP.

Всички тези компоненти са обвързани помежду си и изграждат цялостната инфраструктура на уеб услугите. Ще се спрем по-подробно на всеки един от тях.

Директории за уеб услуги

Директориите за уеб услуги представляват единно място, където различни производители публикуват информация за услугите, който предлагат. Директориите са като уеб указател за услуги (каталог). Самите уеб услуги са организирани в различни категории, като по този начин е улеснено намирането на услуги за определена цел или поставена задача. Директориите предлагат търсене на услуги по зададени параметри (производител, категория, име). Публикуването и търсенето на информация за дадена уеб услуга става посредством UDDI стандарта.

Universal Description, Discovery, and Integration (UDDI)

UDDI представлява отворен XML базиран стандарт за регистриране, откриване и свързване към уеб услуги. Спецификацията му е разработена първоначално съвместно от Microsoft и IBM, а в момента се поддържа и развива от консорциума OASIS (Organization for the Advancement of Structured Information Standards).

UDDI сам по себе си също е уеб услуга. Нейната функционалност включва регистрация и търсене на други услуги.

Microsoft предлага набор от класове (UDDI SDK), чрез които могат да се разработват приложения, използващи цялата мощ и гъвкавост на UDDI. Тези класове напълно съвпадат с описаните в спецификацията на UDDI стандарта.

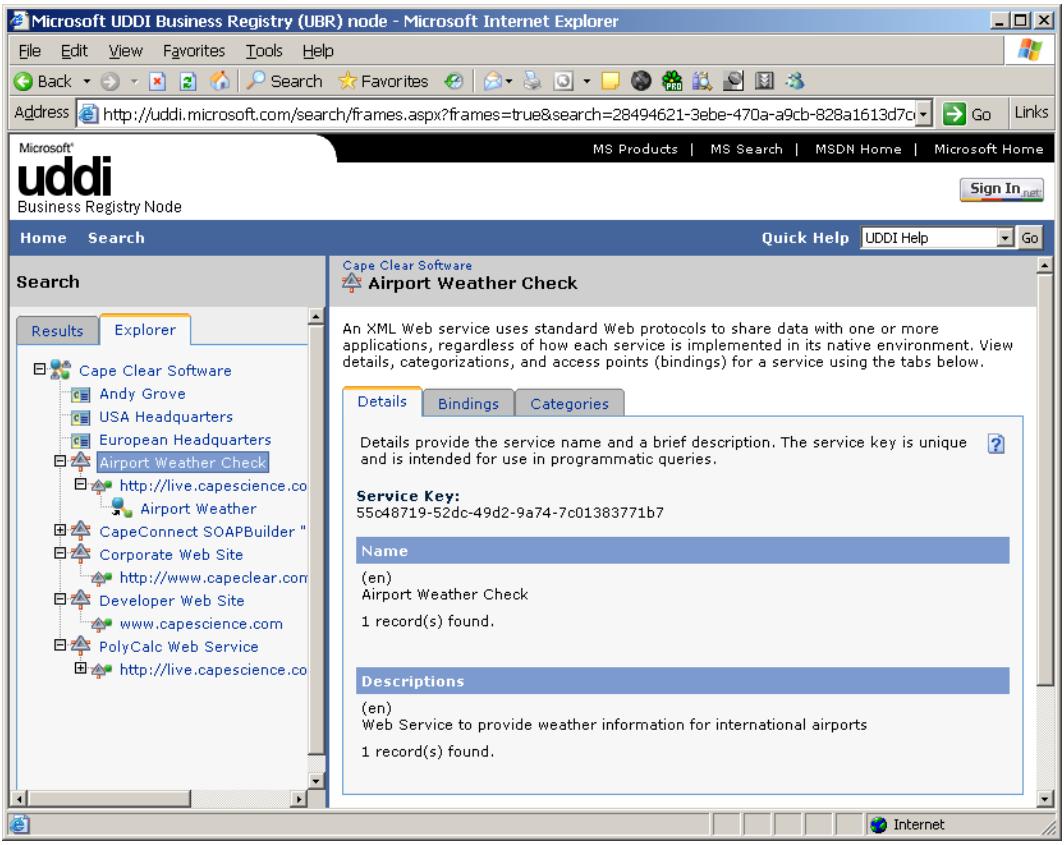
Примери за UDDI директории

Ето няколко примера за UDDI директории, публично достъпни в Интернет:

- http://uddi.microsoft.com (http://test.uddi.microsoft.com)
- https://uddi.ibm.com/ubr/registry.html (https://uddi.ibm.com/testregistry/registry.html)
- http://uddi.sap.com (http://udditest.sap.com)

Забележка: в скоби са адресите, които могат да се използват за тестови цели или по-време на разработка.

Ето как изглежда UDDI директорията на Microsoft:



На картинката можем да видим, че сме намерили уеб услуга, която ще ни предостави информация какво е времето в голяма част от международни-

те летища. Освен тази услуга компанията-производител (Cape Clear Software) предлага още няколко, които можем да видим в лявата част на прозореца. В детайлите за услугата се вижда и нейният **Service Key**. В UDDI регистрите при регистрирането на дадена уеб услуга, на нея й се дава уникален идентификатор – **Service Key**, който я прави уникална в глобалната мрежа. Този идентификатор след това може да се използва за динамичен достъп до услугата през UDDI.

Откриване на уеб услуги

Разгледахме какво представляват UDDI директориите за услуги, а сега ще проследим процеса на откриване на уеб услуги. Този процес има за цел локализация и извличане на описанията на уеб услуги, разположени върху даден сървър. За откриването на услугите се използва DISCO спецификацията.

DISCO (Discovery of Web Services)

DISCO спецификацията е разработена от Microsoft и името и идва от нейното предназначение. DISCO е също XML базиран стандарт и има изключително проста структура. Неговата цел е да посочи връзката към файла с описанието на уеб услугата, връзката към самата услуга и връзка към документация за услугата. Документите с DISCO описание се съхраняват във файл с разширение **.disco**. Ето пример за такъв файл:

TypesService.disco

```
<?xml version="1.0" encoding="utf-8"?>
<discovery
  xmlns="http://schemas.xmlsoap.org/disco/">
    <contractRef
      ref="http://www.myserver.com/Demo-5-Service-
       Types/TypesService.asmx?wsdl"
      docRef="http://www.myserver.com/Demo-5-Service-
       Types/TypesService.asmx"
      xmlns="http://schemas.xmlsoap.org/disco/scl/" />
    <soap
      address="http://www.myserver.com/Demo-5-Service-
       Types/TypesService.asmx"
      xmlns:q1="http://www.myserver.com/schemas/
       Demo_5_Service_Types/"
      binding="q1:TypesServiceSoap"
      xmlns="http://schemas.xmlsoap.org/disco/soap/" />
</discovery>
```

Обикновено **.disco** файлът се разполага в главната директория на съответната уеб услуга. Например, ако услугата е разположена на уеб сървъра www.myserver.com и се казва **math** и главната й директория е със същото име, то пътят към **.disco** файла за тази услуга ще бъде http://www.myserver.com/math/math.disco.

Уеб услугите, разработени с ASP.NET, връщат своя **disco** файл, когато са извикани с параметър **?disco**, например http://www.myserver.com/Demo-5-Service-Types/TypesService.asmx?disco.

UDDI и DISCO

Както вече разбрахме, и UDDI и DISCO ни предлагат начини за откриване на уеб услуги. А каква е разликата между двата стандарта?

UDDI ни предоставя централизирано място регистриране на услуги, където те са разпределени в различни категории, спрямо някакви признаци. Ако искаме нашата уеб услуга да е обществено достъпна и лесно откриваема, трябва да я регистрираме в UDDI регистрите.

Стандартът DISCO ни дава възможност за откриване на всички уеб услуги, разположени локално на конкретен сървър, поради което е много удобен в процеса на разработката на софтуер. DISCO е и тясно интегриран с Visual Studio .NET.

Някои уеб услуги може да нямат DISCO описание, защото то не е задължителна, а само препоръчителна част от уеб услугите.

WSDL описания на услуги

Следващата важна част от инфраструктурата на уеб услугите е тяхното описание. За да можем да използваме дадена уеб услуга, трябва да знаем нейния интерфейс за достъп. Той представлява описание на методите, които уеб услугата предоставя, техните имена, входни и изходни параметри, използваните типове данни и други метаданни за самата услуга.

За описанието на интерфейса и начина на достъп до уеб услуги се използва XML базираният език WSDL (Web Services Description Language), чете се "уиздъл". Той също е отворен стандарт и с неговото развитие се занимава W3C (World Wide Web Consortium).

Едно от предимствата на WSDL е, че той е разширяем във всяко едно отношение. Той нито ни обвързва с конкретен транспортен протокол, нито изисква определена схема, по която да описваме сложните типове, които използваме в нашите услуги. Така е възможно различни технологии да използват различни XML базирани описания на типовете данни, с които работят.

Също както връщат своя **disco** файл, уеб услугите разработени с ASP.NET, връщат WSDL описанието си когато бъдат извикани със специалния параметър **?wsdl**. Например:

http://www.myserver.com/Demo-5-Service-Types/TypesService.asmx?wsdl

WSDL описание – пример

Нека да разгледаме по-подробно WSDL описанието на една примерна уеб услуга – услугата **TypesService.asmx** (вж. примера от точка "Прехвърляне на типове"):

TypesService.wsdl

```
<?xml version="1.0" encoding="utf-8"?>
<definitions
   xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:s="http://www.w3.org/2001/XMLSchema"
   xmlns:s0="http://www.myserver.com/Demo_5_Service_Types/"
   xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
   xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
   targetNamespace=
     "http://www.myserver.com/services/Demo_5_Service_Types/"
   xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <s:schema
     elementFormDefault="qualified" targetNamespace=
          "http://www.myserver.com/Demo_5_Service_Types/">
      <s:import namespace="http://www.w3.org/2001/XMLSchema" />
      <s:element name="GetColors">
        <s:complexType />
      </s:element>
      <s:element name="GetColorsResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1"
              name="GetColorsResult" type="s0:ArrayOfColor" />
          </s:sequence>
        </s:complexType>
      </s:element>
      ...
    </s:schema>
  </types>
  <message name="GetColorsSoapIn">
    <part name="parameters" element="s0:GetColors" />
  </message>
  <message name="GetColorsSoapOut">
    ...
  </message>
  ...
  <portType name="TypesServiceSoap">
    <operation name="GetColors">
       <documentation>
         Returns a list of available colors.
       </documentation>
      <input message="s0:GetColorsSoapIn" />
      <output message="s0:GetColorsSoapOut" />
    </operation>
    <operation name="CalculateDistance">
      ...
```

```
    </operation>
     ...
  </portType>
  <binding name="TypesServiceSoap" type="s0:TypesServiceSoap">
    <soap:binding
       transport="http://schemas.xmlsoap.org/soap/http"
       style="document" />
    <operation name="GetColors">
      <soap:operation soapAction="http://www.devbg.org/services/
           Demo_5_Service_Types/GetColors"
           style="document" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
    </operation>
    ...
  </binding>
  <service name="TypesService">
     <documentation>
       Demo Web service - demonstrates complex type marshalling.
     </documentation>
    <port name="TypesServiceSoap" binding="s0:TypesServiceSoap">
      <soap:address location="http://www.myserver.com/Demo-5-
          Service-Types/TypesService.asmx" />
    </port>
  </service>
</definitions>
```

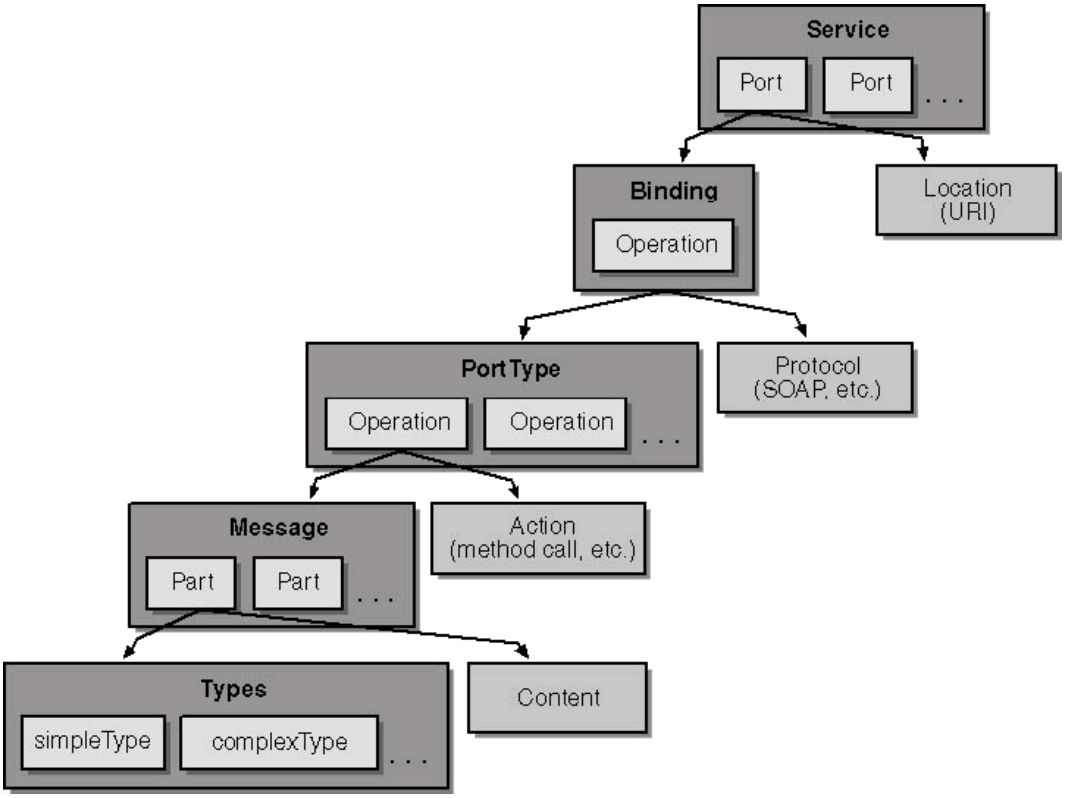
Структура на WSDL описанието

WSDL изгражда едно абстрактно описание на услугата – поддържаните методи, използваните типове данни, начин за достъп до услугата, както и самият й адрес. По-важни елементи от WSDL описанието са:

- **types** – типът на данните, които се подават или връщат от услугата. По подразбиране се използва XSD за описването им (вж. темата "Работа с XML", но може да се ползва и друга схема за описание на данни.
- **message** – описва съобщенията, които ще се обменят между услугата и клиента, както и от какъв вече описан тип са параметрите им.
- **portType** – абстрактната дефиниция за уеб услуга. Състои се от конкретни операции (методи). Спрямо вида на операцията може да има входни, изходни или съдържащи грешка елементи (fault), които сочат към вече описаните съобщения.

- **`binding`** – указва по какъв начин ще се обменят съобщенията за всяка от вече дефинираните операции.
- **`service`** – конкретната дефиниция на уеб услугата, която групира всички вече описани в **`PortType`** операции с конкретен **`Binding`**. Задава и адреса на уеб услугата.

Ето и диаграма, която представя описаните връзки между отделните елементи на WSDL описанието:



SOAP – формат на заявките

След като разгледахме как се откриват и описват уеб услугите, сега ще се спрем на най-важната част от тяхната инфраструктура, а именно как те се пренасят в хетерогенна среда каквато е Интернет.

В ядрото на уеб услугите стои SOAP стандартът, благодарение на който клиентът и доставчикът на услуги обменят съобщения помежду си.

SOAP (Simple Object Access Protocol) е XML базиран формат за обмяна на структурирана и типизирана информация в уеб пространството. При създаването на стандарта е спазван един основен принцип – той да не бъде усложняван допълнително (сложни архитектури като CORBA, вече са показали, че не са ефективни в Интернет), от тук идва и неговото име.

Предимства на SOAP

Не само заради своята простота SOAP е получил подкрепата на Microsoft, IBM, Sun Microsystems, SAP и др. Ето някои други негови преимущества:

- Използва вече утвърдени в Интернет модели: XML за описание на съобщението, XSD за описание на използваните типове и HTTP като транспортен протокол.
- Дефинира своя собствена и независима система за обмяна на съобщения (messaging framework), която е гъвкава и лесно разширяема.
- Не е тясно свързан с конкретен език за програмиране или платформа за разработка. SOAP не предоставя програмен интерфейс (API), а оставя неговата разработка за конкретния език или платформа (.NET Framework, Java, PHP, ...).
- Предоставя междуплатформена комуникация, защото в самата си същност той е изграден върху отворени стандарти (XML).
- Не е свързан с конкретен транспортен протокол. Като такъв може да се използва всеки, който може да пренася XML (например HTTP, TCP, FTP, SMTP, ...).
- Стандартно SOAP съобщенията се пренасят по HTTP, което позволява те да бъдат преминават през защитни стени (firewalls).

Развитието на SOAP

В началото на своето развитие (1998 г.) SOAP спецификацията се е свързвала главно с технологията на XML базираните отдалечени извиквания на методи – XML-RPC (Remote Procedure Call). Тя е описвала по свой собствен начин типовете, които са се предавали между клиента и сървъра.

През 1999 г. създателите на SOAP залагат на XML Schema (XSD), чрез която да описват типовете, използвани в SOAP съобщението.

По-късно, през 2001 г., XSD е приет официално от W3C като препоръчван стандарт за обмяна на XML съобщения.

Постепенно SOAP набира популярност и получава подкрепата на Microsoft. Формира се версия 1.0 на стандарта. IBM и други софтуерни компании виждат преимуществата на SOAP и през пролетта на 2001 г. заедно с Microsoft изготвят версия 1.1. Промените спрямо версия 1.0 са незначителни, но по-важното е, че вече целта на SOAP не е единствено отдалеченото извикване на методи. SOAP сам по себе си вече представлява framework за обмяна на XML базирани съобщения. Тогава е предложена пред W3C и приета като официален стандарт спецификацията SOAP версия 1.1.

В момента SOAP стандартът има версия 1.2, като промените спрямо 1.1 не са значителни (една от тях е, че SOAP вече не е акроним, а просто име). В .NET Framework 1.1 е реализиран SOAP версия 1.1.

Структура на SOAP съобщенията

Структурата на едно SOAP съобщение не е сложна. Тя се състои от две части: SOAP хедър (header) и SOAP тяло (body), а като коренов елемент на цялото съобщение стои елементът плик (envelope), който трябва да съдържа пространството от имена (namespace) за SOAP съобщението.

На следващата фигура е показана структурата на едно просто SOAP съобщение. Можем да видим и пространството от имена, което в случая е http://schemas.xmlsoap.org/soap/envelope/ спрямо версия 1.1 на SOAP стандарта. Във версия 1.2 това пространство е сменено и неговият нов идентификатор в глобалната мрежа (URI – Uniform Resource Identifier) е: http://www.w3.org/2003/05/soap-envelope/.

```
<?xml version="1.0" encoding="utf-8"?>

<soap:Envelope
   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

   <soap:Header>
        <From>Stefan</From>
        <To>Dido</To>
   </soap:Header>

   <soap:Body>
        <Task>Make me a cup of coffee.</Task>
    </soap:Body>

</soap:Envelope>
```

След като разгледахме примерното SOAP съобщение, сега ще се спрем по-подробно на основните негови елементи: хедър и тяло.

SOAP хедър

Елементът хедър не е задължителен в едно SOAP съобщение, но ако той присъства задължително трябва да е преди тялото на съобщението. Неговата цел е да разшири самото съобщение като добави допълнителна мета информация (метаданни) извън тялото. Тази информация може да има различен характер. Ето и някои от най-честите приложения на SOAP хедъра:

- Маршрутизация (routing) – ако съобщението е предназначено за няколко получателя или трябва да премине през различни точки, в хедъра може да се укаже информация за маршрутизиране (последователността на преминаване през отделните точки).
- Транзакция – ако съобщението е част от разпределена транзакционна система, тя може да се опише в хедъра.

- Автентикация (authentication) – получателят може да изисква подателят да се автентикира преди съобщението да бъде обработено. Автентикацията може да се базира на пароли, на цифрови подписи и сертификати или на друг механизъм.
- Сигурност – ако получателят иска да е сигурен, че съобщението не е подправено, подателят може да подпише цифрово тялото на съобщението и да постави генерирания подпис в хедъра (технологията на цифровия подпис в описана подробно в темата "Сигурност в .NET Framework".
- Компресия – ако тялото на съобщението е обемно, подателят може да го компресира и да укаже вида на използвания компресиращ алгоритъм в хедъра.

SOAP тяло

За разлика от хедъра SOAP тялото е задължителен елемент, защото в него е разположена същността на съобщението. Тялото може да съдържа какъвто и да е текст (в частност XML), стига неговата структура да не разваля тази на съобщението. Има два вида SOAP съобщения: процедурно-ориентирани (procedure-oriented) и документно-ориентирани (document-oriented).

При документно-ориентираните съобщения всякакъв вид информация може да бъде кодирана (encoded) и изпратена до съответния получател. .NET Framework, например, ни дава възможността и да сериализираме даден обект чрез **`SoapFromatter`** (вж. темата "Сериализация на данни") и да го изпратим като SOAP съобщение.

За разлика от документно-ориентирани съобщения при процедурно-ориентираните се осъществява двустранна комуникация между клиента и сървъра. Клиентът изпраща заявка към сървъра, като в тялото на съобщението указва, кой предоставен метод иска да извика и с какви параметри. Сървърът обработва заявката и връща отговор, съдържащ резултата от изпълнението на метода, обратно към клиента.

Обмяна на SOAP съобщения – пример

Ето примерна заявка, направена от клиент, който извиква метод за намиране на разстоянието между две точки в правоъгълна координатна система:

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <CalcDistance xmlns="http://www.devbg.org/Calc">
      <p1>
```

```
            <x>4</x>
            <y>5</y>
         </p1>
         <p2>
            <x>7</x>
            <y>-3</y>
         </p2>
      </CalcDistance>
   </soap:Body>
</soap:Envelope>
```

Всяка точка е представена като структура с две координати. Методът, който ще се извика на сървъра с параметри двете точки, е **CalcDistance**. Ето и отговорът, който ще се върне след неговото успешно изпълнение:

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
      <CalcDistanceResponse xmlns="http://www.devbg.org/Calc/">
         <CalcDistanceResult>
            8,54400374531753
         </CalcDistanceResult>
      </CalcDistanceResponse>
   </soap:Body>
</soap:Envelope>
```

SOAP грешка (fault)

Тялото на SOAP съобщението може да съдържа и информация за възникнали грешки или изключения при изпълнението на услугата. Информацията за грешката, ако има такава, се записва в елемента **fault**.

Ето тялото на примерно SOAP съобщение със SOAP грешка, което е изпратено към клиента, защото на сървъра е хвърлено изключение:

```
<soap:Body
   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>
         Attempted to divide by zero.
      </faultstring>
      <detail />
   </soap:Fault>
</soap:Body>
```

SOAP fault може да съдържа няколко елемента. Ето по-важните от тях:

- **faultcode** – указва къде е възникнала грешката.
- **faultstring** – предоставя текстово описание на грешката.
- **detail** – незадължителен елемент, който може да съдържа XML с допълнителна информация за грешката.

Протоколен стек на уеб услугите

Уеб услугите, както вече може би сте се убедили, обединяват в себе си много технологии. За всяка отделна част от тяхната инфраструктура има отделен стандарт, който описва как се извършва взаимодействието с останалите части от инфраструктурата. За да изясним взаимоотношенията между отделните стандарти и протоколи, нека разгледаме следващата фигура, която описва отделните слоеве от комуникацията с уеб услугите:

Ще започнем отдолу нагоре, като се движим от по-общите стандарти към тези, който са по-тясно свързани с уеб услугите.

На ниво транспорт стои протоколът HTTP, който е в основата на WWW и уеб технологиите. Както вече знаем, HTTP реализира модела "заявка - отговор", който се използва и при уеб услугите. Един от недостатъците на HTTP е, че не поддържа стандартно сесия, т. е. той е stateless (няма състояние). Всяка заявка е отделна за себе си и независима от останалите. По тази причина уеб услугите най-често реализират функционалност, при която не се пази състояние, но чрез специални техники, които ще разгледаме по-нататък, този проблем може да се преодолее.

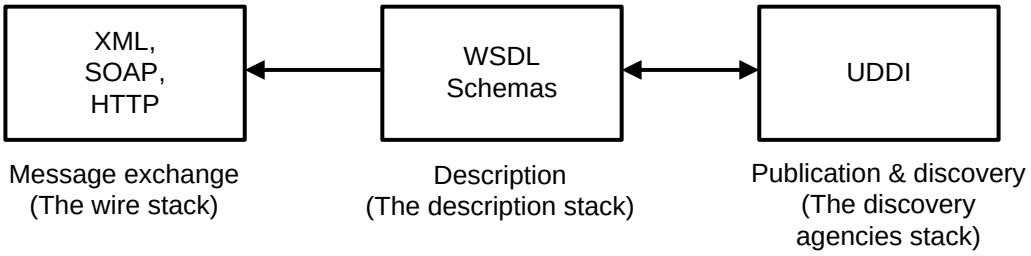
Основното предназначение на HTTP е да пренася данни от уеб сървъра към клиента. В случая на уеб услугите тези данни са под формата на XML. Със своята простота, мощ и разширяемост XML е най-универсалният начин за пренасяне на структурирана информация в глобалната мрежа. Именно поради това, XML е в основата на всички стандарти и протоколи, свързани с уеб услугите.

Описанието на структурата и формата на предаваните във вид на XML данни се извършва чрез XSD схеми. Чрез XSD се описват типовете данни, пренасяни при извикването на уеб услуги.

Чрез вече описаните типове се изгражда абстрактният програмен интерфейс на услугата чрез езика за описание на уеб услуги (WSDL).

SOAP стандартът осигурява инфраструктура за изпращане и получаване на XML съобщения, чрез които се извикват предоставените методи от уеб услугата и се получава резултатът от тяхното изпълнение.

Ето и една диаграма, която показва взаимовръзките между отделните протоколи и стандарти, разделени в три логически стека: стек за обмяна на съобщения, стек за описания и стек за откриване на уеб услуги:



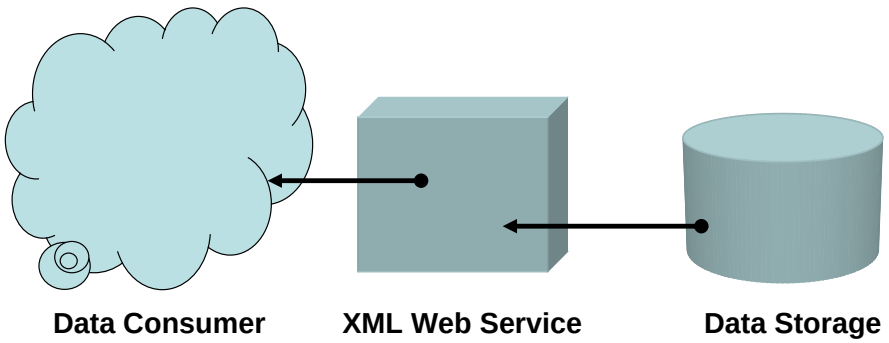
Сценарии за използване на уеб услугите

След като се запознахме с архитектурата и инфраструктурата на уеб услугите, сега нека разгледаме различни сценарии, в които можем да използваме уеб услуги.

Уеб услугите намират все по-голямо приложение при изграждането на разпределени системи, защото са слабо обвързани с клиента, използват отворени стандарти и на практика са "почти" универсални. Ето някои от най-често срещаните сценарии, в които уеб услугите играят важна роля.

Доставяне на данни

Едно от най-честите приложения на уеб услугите е за реализиране на достъпа до база данни (или информация съхранена на отдалечено място). Единствената роля на уеб услугата в този случай е да се свържи с доставчика на данните, да ги извлече и после да ги предостави на клиента. Следната диаграма илюстрира този сценарий:



Услуги към клиентски приложения

С нарастване на употребата на уеб услугите, много приложения (в това число и MS Office) предоставят възможност за използване и консумиране на услуги директно. По този начин се избягва изграждането на междинен слой между услугата и клиентското приложение, а това подобрява вътрешната структура на приложението. С масовата употреба на уеб услуги, на която сме свидетели, се очаква все повече настолни и сървърни приложения да се възползват от тяхната мощ.

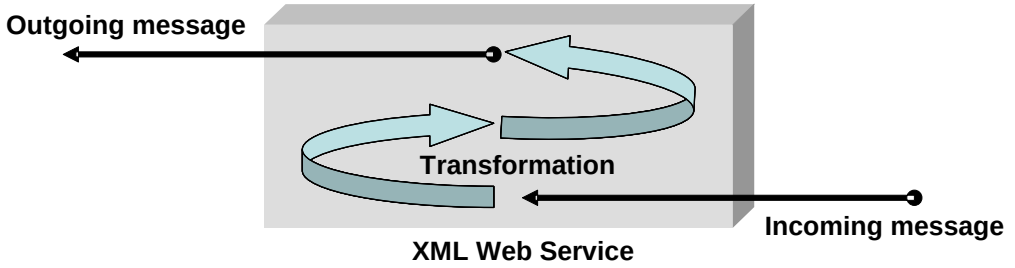
Интеграция на приложения

Друг много често срещан сценарий за използването на уеб услуги е за интеграция между различни приложения. Самата интеграция може да е:

- Бизнес интеграция – изграждане на бизнес процес, базиран на уеб услуги, чиито консуматори ще са отделните компании, участващи в процеса. Пример: предаване на поръчка от продавача към този, който е отговорен за нейната доставка.
- Междуплатформена интеграция – взаимодействие между приложения работещи на различни платформи под различни операционни системи. Пример: уеб услуга, разработена на ASP.NET, се консумира от мобилно клиентско приложение, разработено на Java.

В ролята на адаптери

При някои приложения уеб услугите могат да влизат в ролята на адаптери, т. е. да трансформират по зададени правила и схеми входящото съобщение и да го препращат по веригата към неговия получател.



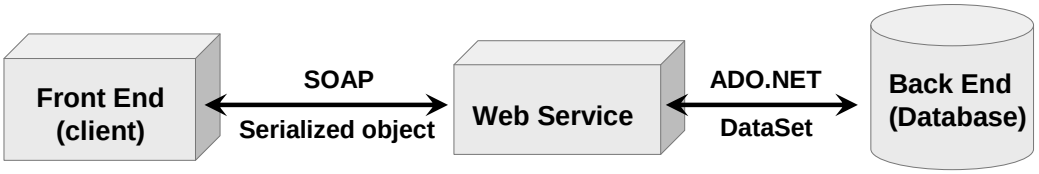
Такова приложение на уеб услугите не е масово разпространено, но успешно се използва в редица големи софтуерни проекти. Използването на уеб услугите като трансформиращи адаптери е концепция, успешно имплементирана в Microsoft BizTalk Server, където има отделен framework за разработка на адаптери.

Връзка между отделните компоненти на Enterprise приложения

В последните години се налага тенденцията различните компоненти на едно Enterprise приложение (ще разгледаме по-подробно Enterprise приложенията след малко) да комуникират помежду си чрез уеб услуги.

Голяма част от новите продукти на Microsoft освен стандартен потребителски интерфейс (уеб приложение или Windows desktop приложение) предлагат и уеб услуга, чрез която разработчиците могат да използват и надграждат приложението (уеб услугата представлява API за достъп до основната функционалност).

Следващата картинка илюстрира използването на услуга за връзка към базата данни в трислойно приложение. В него чрез класовете от ADO.NET за достъп до базата данни се извличат данните под формата на `DataSet` обекти. Уеб услугата обработва тези данни и от създава бизнес обекти, които се изпращат към клиента чрез SOAP съобщения:



Enterprise приложения

След като разгледахме най-честите приложения на уеб услугите, сега ще се спрем по-подробно на Enterprise приложенията и ще видим защо уеб услугите играят ключова роля при тяхното изграждане.

Кои приложения са Enterprise?

"Enterprise приложенията" означава многослойни разпределени приложения, които отговарят на изискванията на големите корпоративни клиенти. Те се състоят от множество компоненти, които са интегрирани помежду си и работят като едно цяло. За да отговарят на съвременните бизнес изисквания и стандарти, тези разпределени приложения трябва да имат следните характеристики:

- Изключително надеждни (reliable) – една грешка в неподходящ момент може да причини огромни загуби.
- Силно скалируеми (scalable) – да поемат и обработват заявките на стотици потребители, които работят конкурентно в даден момент.
- Лесно разширяеми (extensible) – когато клиентът поиска нова функционалност да не се налага цялото приложение да бъде пренаписано, а само да се разшири неговата функционалност.

- Сигурни (secure) – сигурността в приложението да не е "закърпена", а да е залегнала дълбоко в неговия дизайн.
- Устойчиви на сривове (fault tolerant) – да работят в критични ситуации, а когато някой от компоненти на приложението спре да работи, това да не води до срив на цялата система.

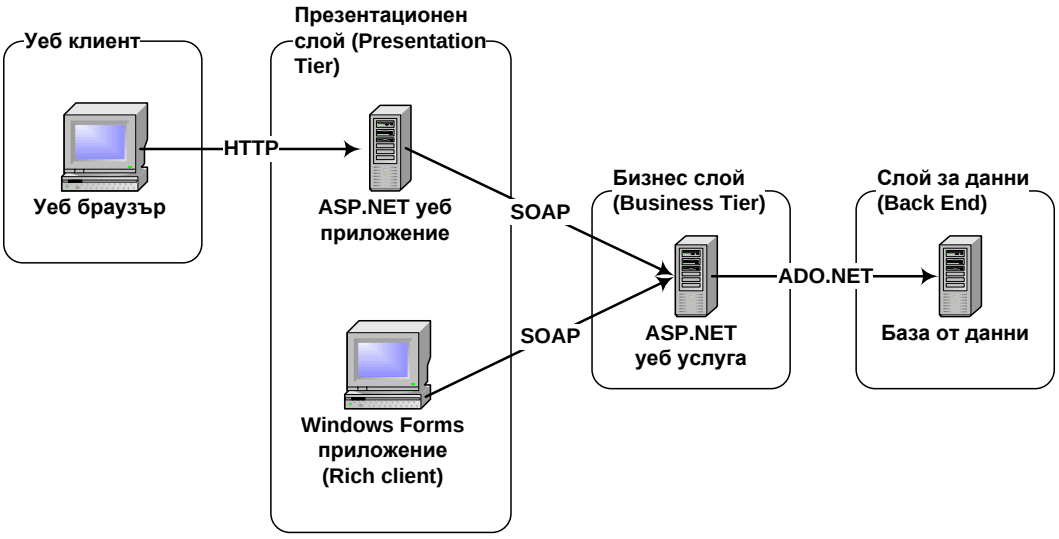
.NET Enterprise приложения

По-принцип Enterprise приложенията са многослойни, но класическата архитектура за тях си остава трислойната: слой за данните, бизнес слой и презентационен слой. Няма да се спираме подробно на тази архитектура, защото подробно вече разгледахме нейните характеристики в темата "Достъп до данни с ADO.NET". Ще разгледаме само изграждането на бизнес слоя чрез уеб услуги.

Бизнес логика в уеб услуги

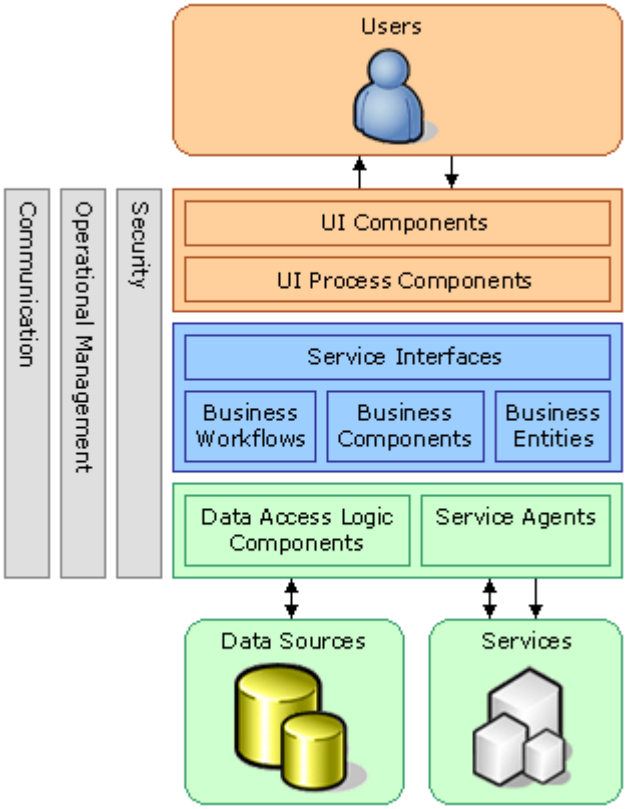
В съвременните системи все по-често цялата бизнес логика на приложението бива изнесена в уеб услуги. Това е така, защото уеб услугите са лесно достъпни през Интернет и осигуряват възможност за междуплатформена комуникация, т. е. техните консуматори може да са много различни: уеб приложения, Windows Forms клиенти, мобилни устройства, както и други смарт клиенти (smart clients). Уеб услугите отварят системата към взаимодействие с различни крайни клиенти, реализирани върху различни платформи.

На следващата диаграма е показана схематично класическата трислойна архитектура, реализирана със средствата на .NET платформата с използването на уеб услуги:



Разпределени приложения (Distributed Applications)

Както вече знаем, многослойните разпределени приложения също спадат към групата на Enterprise приложенията. Многослойните приложения са по-комплексни от трислойните, но при тях скалируемостта е по-голяма, защото отделните компоненти могат да се разположат на различни сървъри и да се оптимизират поотделно (разпределянето, върху няколко сървъра, на отделни уеб услуги или компоненти от едно приложение, се нарича "уеб ферма" – Web farm). Следващата диаграма показва визията на Microsoft за изграждането на разпределени приложения:



Да започнем разглеждането на диаграмата отгоре надолу. Най-отгоре стои потребителят, който вижда единствено потребителския интерфейс за работа с приложението. Той не се интересува нито каква база от данни използваме, нито от колко слоя е изградена архитектурата; за него е важно колко е бързо и надеждно е приложението.

Препоръчва се да няма никаква вградена (hard-coded) информация в компоненти, който осъществяват UI (User Interface), а всякаква логика по навигацията да се изнесе в отделни компоненти (UI Process Components).

Следва бизнес слоят, който е разделен на четири компонента:

- Business Workflow – реализира бизнес превилата, които се прилагат в системата и извършва оркестрацията между отделните бизнес процеси (осъществява свързването на процесите);
- Business Components – реализират самата бизнес логика на приложението (основните работни процеси);
- Business Entities – представляват модели на бизнес обекти от реалния свят (например продукт, клиент, поръчка, ...);
- Service Interface – уеб услуги, които предоставят достъп до бизнес логиката на приложението на най-високо ниво. Те представляват програмното API на приложението (т. нар. бизнес фасада).

На най-ниско ниво са разположени компонентите за достъп до базата (Data Access Logic Components) и компонентите, които указват как външни услуги могат да използват тези предоставени от системата. Паралелно с цялата тази структура върви и предоставената ни от .NET Framework богата функционалност за управление на сигурността и изключенията.

Уеб услугите в ASP.NET

.NET Framework ни дава богата инфраструктура и множество от стандартни класове, чрез които лесно и бързо се създават и използват уеб услуги. Те имплементират предаването и приемането на SOAP съобщения, осигуряват преобразуването на типове от XML в .NET типове и обратно, предоставят възможност за автоматично генериране на WSDL описания и автоматично генериране на прокси класове от WSDL описания. Благодарение на ASP.NET сложната инфраструктура, свързана с използването на уеб услугите остава скрита за програмиста.

Нека разгледаме в детайли средствата на .NET платформата и по-специално на ASP.NET за работа с уеб услуги.

Пространства от имена

В ASP.NET има няколко пространства от имена, които са свързани със създаването и консумирането на уеб услуги. Нека ги разгледаме накратко.

System.Web.Services

Уеб услугите се реализират в пространството **`System.Web.Services`**. То съдържа всички класове, които са необходими за създаването на уеб услуги чрез .NET Framework. Когато се използва Visual Studio .NET повечето класове на **`System.Web.Services`** остават невидими за разработчика, затова няма да се задълбочаваме в подробно описание на всичко. Трите основни подпространства от имена на **`System.Web.Services`** са **`Description`**, **`Discovery`** и **`Protocols`**.

System.Web.Services.Description

Пространството от имена **System.Web.Services.Description** съдържа класовете, нужни за описанието на уеб услугите, като се използва Microsoft SDL (Service Definition Language) – имплементация на Microsoft на WSDL стандарта.

Visual Studio .NET използва тези класове за да създаде **.disco** и **.vsdisco** файлове. Един от по-интересните класове е **ServiceDescription**. Той ни позволява четен, пишем и обработване WSDL документи. Ето кратък пример за неговото използване:

```
ServiceDescription newDescription =
   ServiceDescription.Read("SomeXMLDescriptionFile.wsdl");
// Manipulate the description or create new Web service using it
newDescription.Write("newService.wsdl");
```

System.Web.Services.Discovery

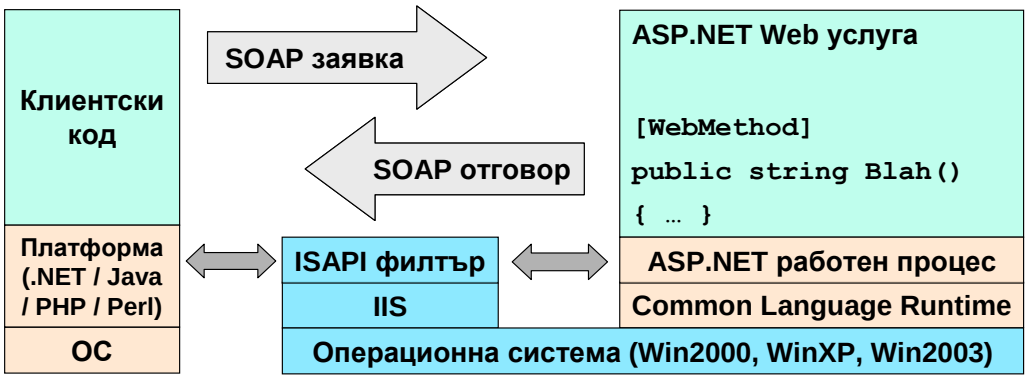
Пространството **System.Web.Services.Discovery** се състои от класовете, които се грижат за откриването на уеб услуги. Във Visual Studio .NET, когато се добави Web Reference, тези класове откриват **.vsdisco** файловете, които описват уеб услугите.

System.Web.Services.Protocols

Пространството **System.Web.Services.Protocols** съдържа класове, които се използват за дефиниране на протоколи, позволяващи преноса на съобщения между ASP.NET уеб услугата и приложенията, които я използват. Тези класове се използват обикновено в така наречените междинни (proxy) класове. Основно се грижат за оформянето на SOAP съобщенията.

Архитектура на ASP.NET уеб услугите

Подобно на ASP.NET приложенията, уеб услугите се разполагат върху уеб сървър (IIS или някой друг) и се изпълняват от работния процес на ASP.NET. Следващата фигура илюстрира как протича процесът на изпълнение на една уеб услуга:



Клиентският код извиква метод на уеб услугата чрез изпращане на SOAP заявка. Уеб услугата изпълнява извикания метод и връща резултата му отново като SOAP съобщение. Данните, които се предават между услугата и приложението, се сериализират като XML.

Уеб услугите използват отворени стандарти и благодарение на това клиентският код може да е разположен върху различни платформи и операционни системи (.NET, Java, PHP, Perl и други).

Като междинен слой стоят ISAPI филтър и уеб сървър (най-често IIS). ISAPI (Internet Server Application Program Interface) позволява на разработчиците да реализират уеб базирани приложения, работещи много по-бързо от стандартните CGI (Common Gateway Interface) приложения. Причината за това се крие в тясната интеграция на ISAPI с уеб сървъра. Освен Internet Information Server на Microsoft има и други сървъри поддържащи ISAPI филтри.

IIS (Internet Information Server) приема HTTP заявките и ги предава на ISAPI, който ги предава на ASP.NET работния процес, който ги обработва.

От страната на уеб услугата стоят ASP.NET работният процес и CLR (Common Language Runtime), които управляват нейното изпълнение.

ASP.NET работният процес е специален процес, който е част от .NET Framework и се грижи за обработката на заявки към ASP.NET. Използва се както при уеб приложения, така и при уеб услуги.

Създаване на уеб услуги

За да се създаде уеб услуга в .NET Framework се създава файл с разширение **.asmx**. За да се укаже, че този файл е уеб услуга, в началото на файла се поставя следният таг:

```
<%@ WebService Language="C#" Class="SomeServiceClass" %>
```

След това се създава клас с име съответстващо на зададеното в тага. Този клас трябва да наследява **System.Web.Services.WebService** или към него да е приложен атрибутът **[WebService]**.

В зависимост каква е целта на услугата, може да се имплементират един или няколко уеб метода, като пред всеки се поставя атрибутът **[WebMethod]**. Този атрибут указва, че даденият метод трябва да е публично достъпен през интерфейса на уеб услугата.

```
[WebMethod]
public void SomeMethod(…)
{
   // Some Code
}
```

Създаване на уеб услуги – пример

Като реален пример ще създадем уеб услуга с единствен уеб метод, който по зададени две цели числа пресмята и връща сумата им.

Създаваме файла **AddService.asmx** и в него записваме следния код:

```
<%@ WebService Language="C#" Class="AddService" %>

using System;
using System.Web.Services;

public class AddService : WebService
{
   [WebMethod]
   public int Add(int a, int b)
   {
      return a + b;
   }
}
```

Уеб услугите и уеб приложенията

Както вече споменахме, уеб услугите в .NET Framework се изпълняват от работния процес на ASP.NET като обикновени уеб приложения. Сходството между уеб услугите и уеб приложенията не спира само до процеса, който ги изпълнява.

Уеб услугите, също както уеб приложенията, могат да се конфигурират и настройват посредством файловете **Web.config** и **Global.asax**. Те също се разполагат върху уеб сървър (обикновено IIS, но не задължително).

Всъщност уеб приложенията приемат HTTP заявки и отговарят с HTTP отговори, чрез които връщат най-често HTML документ. Също като тях, уеб услугите приемат HTTP заявки и отговарят с HTTP отговори, но заявките и отговорите съдържат SOAP съобщения. Сходството е голямо. Разликата е само в съдържанието на заявките, в начина на тяхната обработка и във връщания резултат.

Публикуване на уеб услуги

Публикуването на уеб услуги може да стане по няколко начина, като те по своята същност правят едно и също.

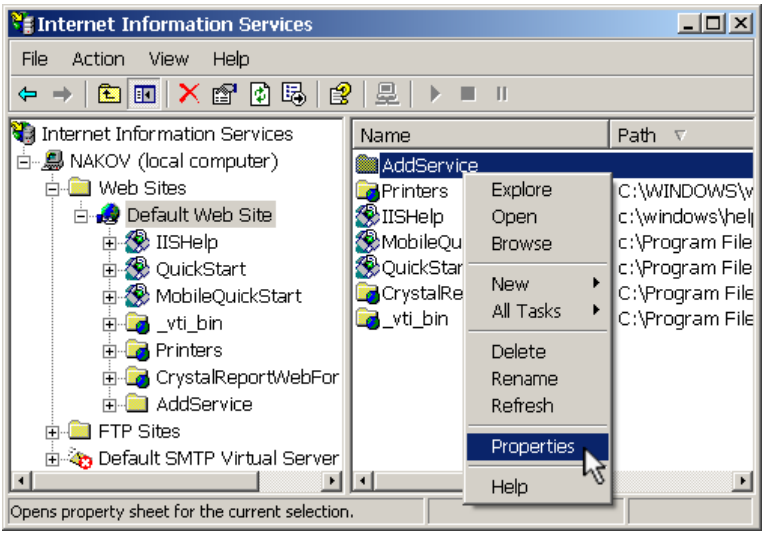
Копиране на услугата в IIS и регистрация

Първият начин за публикуване на уеб услуга е чрез копиране на услугата в IIS и регистрацията ѝ като уеб приложение:

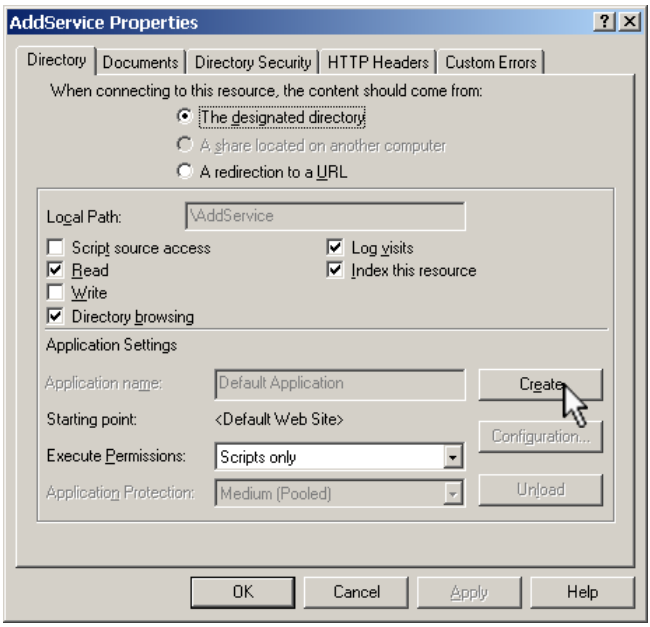
1. Копираме цялата папка на уеб услугата във физическата папка, към която сочи уеб сайтът по подразбиране на IIS (обикновено това е директорията "**C:\Inetpub\wwwroot**").

2. Стартираме административната конзола на IIS. Това можем да направим като от Старт менюто на Windows изберем **Run** и след това в появилия се прозорец напишем "**`inetmgr`**". След като стартира, административната конзола показва всички папки намиращи се в "**`C:\Inetpub\wwwroot`**" и всички останали виртуални директории.

3. Намираме току-що копираната папка **`AddService`** и от контекстното меню избираме **Properties**:



4. Отваря се прозорецът за управление на настройките за съответната папка. За да стане достъпна уеб услугата от тук създаваме Web Application като натиснем бутона **`[Create]`**.



Така нашата първа уеб услуга става достъпна от следния адрес: **http://localhost/AddService/AddService.asmx**.

Регистрация на услугата без копиране в IIS

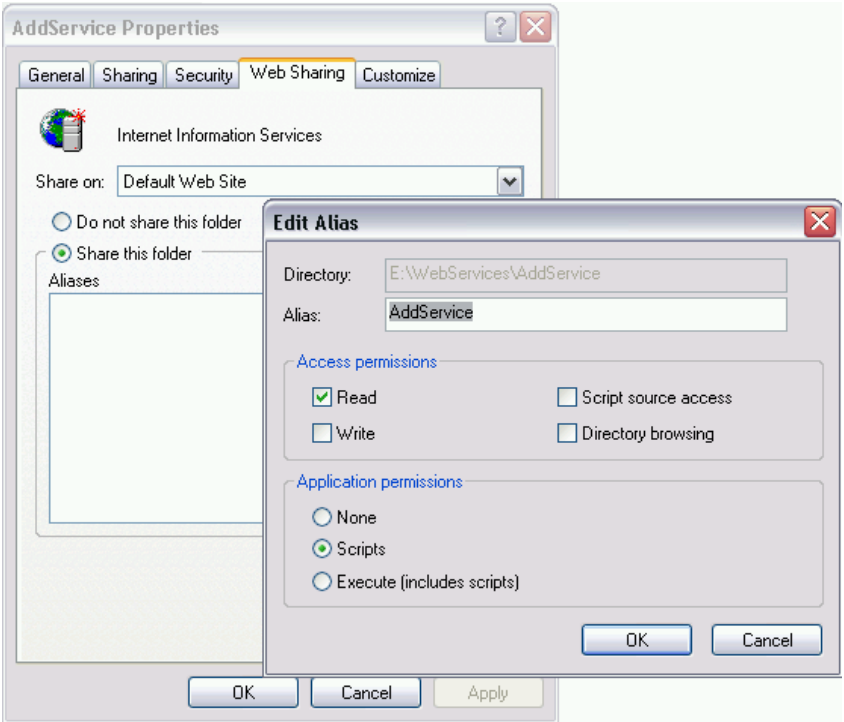
Вторият начин за публикуване, както вече споменахме, по същество прави същото – създава Web Application в уеб сървъра, но го прави по различен начин.

В представената по горе последователност се наложи да копираме папката на услугата в общата директория на IIS. Това по принцип не е проблем, но повечето разработчици предпочитат да имат добра подредба на съдържанието на твърдия си диск. Ако всички уеб приложения се поставят в една папка, това внася малък хаос и води до загуба на време в търсене при по-голям брой приложения. Друго основание, да не се копират уеб услуги или уеб приложения в папката **`wwwroot`**, се появява при уеб сървъри, в които има десетки, дори стотици уеб сайтове и всеки е със собствен URL адрес. В такава една ситуация би било пагубно всички сайтове да са в една папка.

Нека предположим, че физически нашата уеб услуга се намира в папката **`E:\WebServices\AddService`**.

За да създадем уеб приложение за тази услуга се използва един от следните начини:

Регистрация на виртуална директория от Windows Explorer



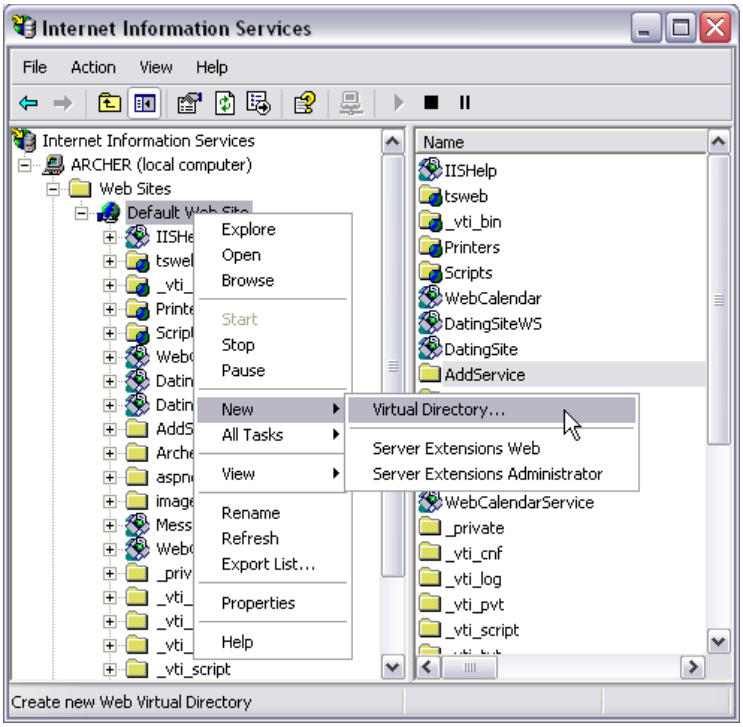
1. Отиваме до папката **`E:\WebServices`** и щракаме с десния бутон на мишката върху папката **`AddService`**.

2. От контекстното меню избираме **Properties**. В отворилият се прозорец отиваме на етикета **Web Sharing**.

3. Избираме **Share this folder** и в резултат се появява прозорец, в който записваме името, което искаме да има нашата уеб услуга. След това натискаме бутона **`[OK]`** за да се върнем обратно в **Properties** прозореца.

Регистрация на виртуална директория от административната конзола на IIS

1. Отваряме административната конзола за управление на IIS. Щракаме с десния бутон на мишката върху **Default Web Site**. Посочваме с мишката **New** и от появилото се контекстно меню избираме **Virtual Directory**:



2. Стартира се **Virtual Directory Creation Wizard**, с помощта на който по много лесен начин се създава уеб приложение. Натискаме **`[Next]`** и на следващата стъпка въвеждаме в полето **Alias** името на нашата услуга. На следващата стъпка може да направим две неща. Или натискаме бутона **`[Browse]`** и в появилия се нов прозорец избираме директорията **`E:\WebServices\AddService`**, или направо я въвеждаме в полето **Directory**. На следващата стъпка избираме какви права за достъп ще има до нашата услуга. Тук може да се оставят тези права,

които са по подразбиране. И така стигаме до последната стъпка, където посредством бутона **[Finish]**, завършваме създаването на виртуалната директория.

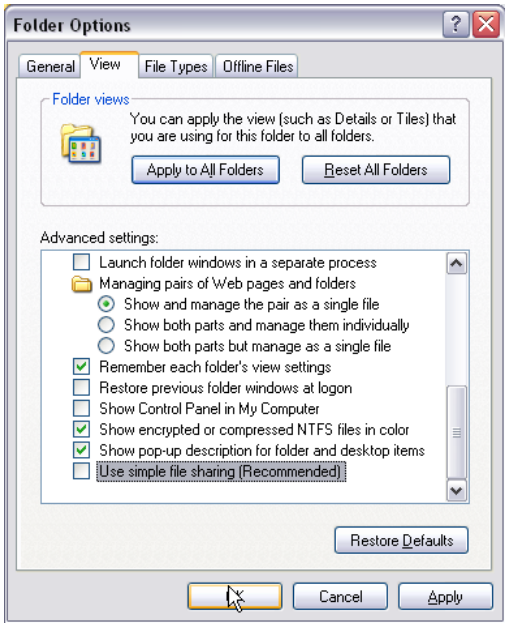
Настройка на правата за достъп

Ако сега се опитаме да отворим адреса на нашата услуга, най-вероятно няма да успеем, поради ограничения на физическия достъп за четене върху папката **E:\WebServices\AddService**. Въпреки, че позволихме достъпа през уеб до адреса **http://localhost/AddService/AddService.asmx**, физическият достъп до съответната папка е забранен.

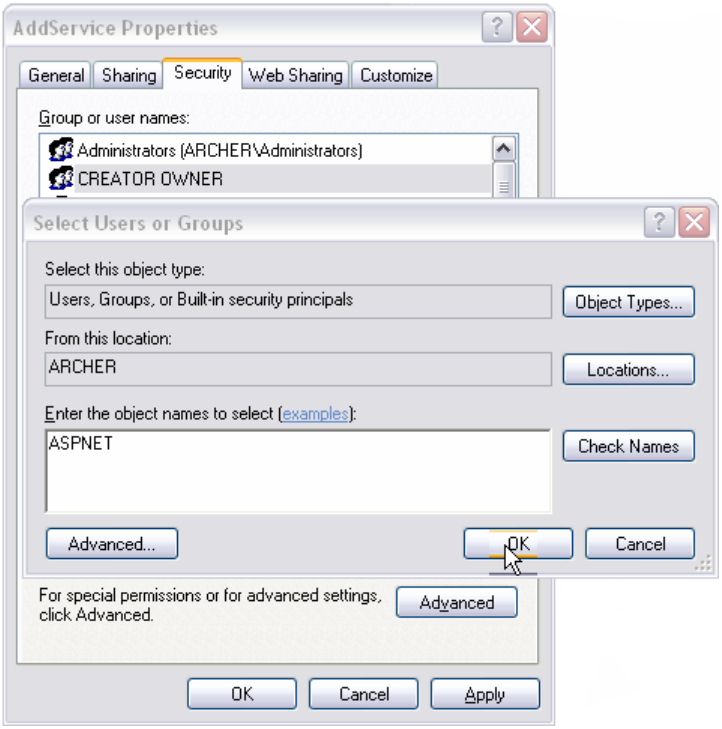
Уеб услугата се изпълнява от работния процес на ASP.NET и когато до уеб сървъра пристигне заявка за дадена уеб услуга, той стартира ASP.NET работния процес. Той пък от своя страна се зарежда с правата за достъп на специалния потребител **ASPNET** (в Windows 2003 Server този потребител се нарича **Network Service**). Така работният процес се опитва да осъществи достъп до физическата папка **E:\WebServices\AddService**, за която обаче няма права.

За да му дадем необходимите права, отваряме Windows Explorer и отиваме до **E:\WebServices**. Щракаме с десния бутон на мишката върху папката **AddService** и от контекстното меню избираме **Properties**. В отворилия се прозорец отиваме на етикета **Security**.

Възможно е този етикет да го няма. Това най-често се случва, когато компютърът не е част от домен. За да го покажем от менюто **Tools** избираме **Folder Options**. В отворилия се прозорец, отиваме на етикета **View**. В частта **Advanced Settings** изключваме последната настройка, а именно **Use simple file sharing**:



И така, вече би трябвало етикетът **Security** да се показва. Отиваме в него и натискаме бутона **[Add]**, в резултат на което се отваря прозорец **Select Users or Groups**. В активното текстово поле въвеждаме потребител **ASPNET** (или за Windows 2003 Server съответно **Network Service**):



С така направените настройки вече уеб услугата би трябвало да работи без проблеми. При първия начин на публикуване избегнахме настройките на правата за физическата папка поради следната причина. При копиране на папката **AddService** в **C:\Inetpub\wwwroot** правата, на потребителя **ASPNET**, за достъп до нея се наследяват автоматично от папката **C:\Inetpub\wwwroot**. Ако обаче за тази папка не са дадени права за достъп, цялата описана по-горе процедура трябва отново да се изпълни.

Компилиране на уеб услугата

Когато създадохме нашата услуга, поставихме целият й сорс код в **.asmx** файл. Поради това не се наложи никаква компилация, но ако кодът е в отделен **.cs** файл, се налага след конфигуриране на услугата, тя да се компилира. В такива случаи **.asmx** файлът се състои единствено от тага:

```
<%@ WebService Language="c#" Codebehind="AddService.asmx.cs"
Class="MyServise.AddService" %>
```

Компилирането на C# кода, който е част от услугата (т. нар. Code Behind), може да се извърши с VS.NET или с конзолния компилатор (например с командата "**csc.exe /target:library**"). Получените при компилацията

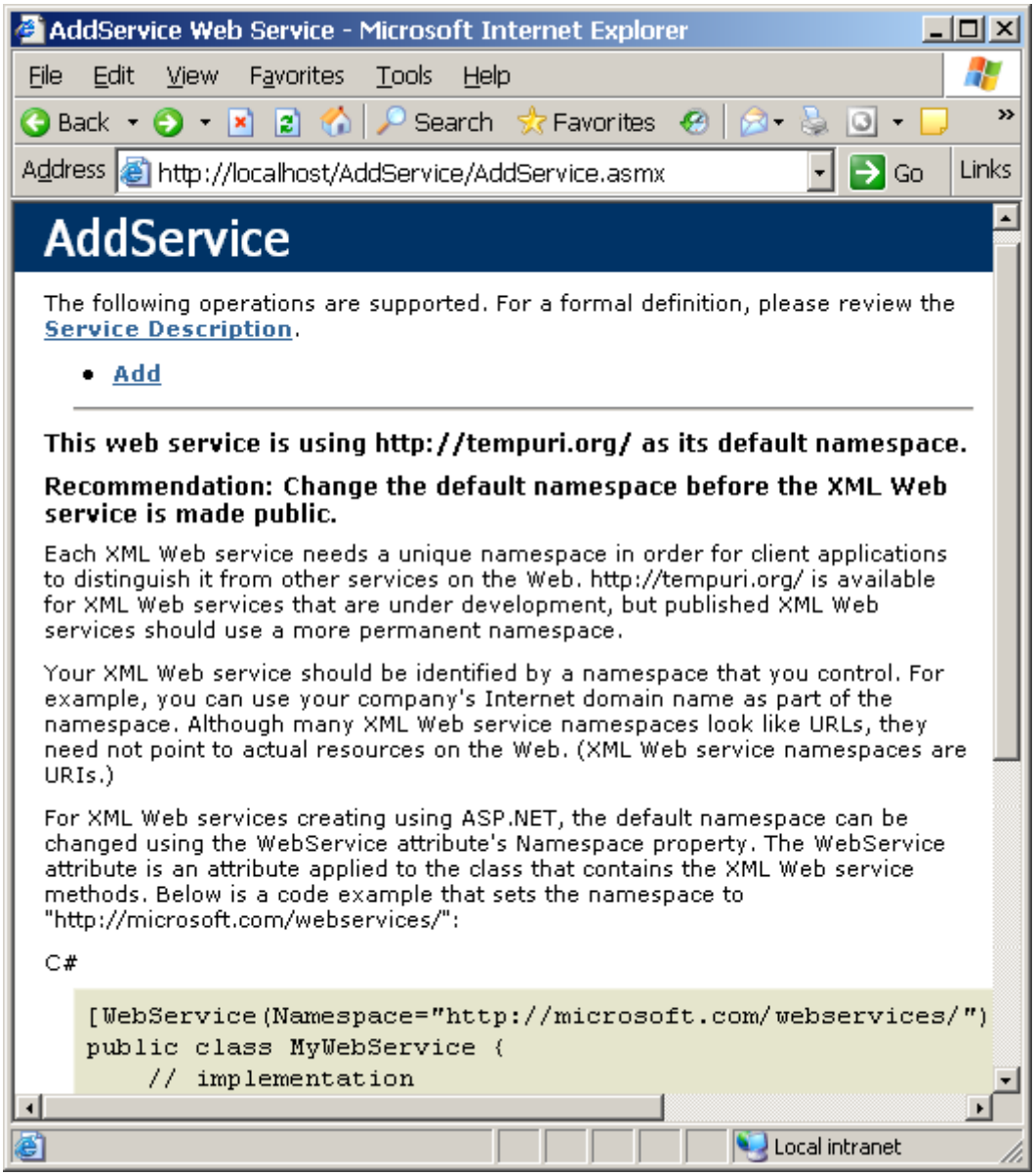
асемблита трябва да се запишат в поддиректория **bin** на виртуалната директория на услугата.

Тестване на нашата първа уеб услуга

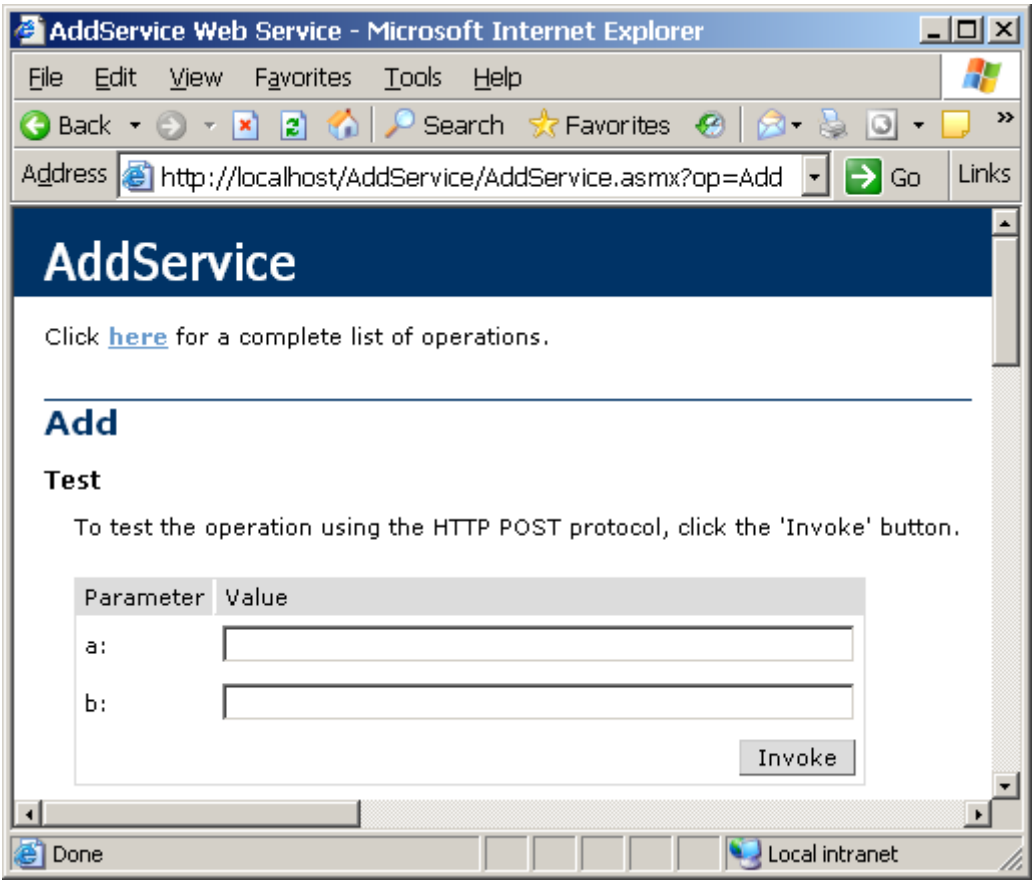
След като вече сме публикували конфигурирали уеб услугата в IIS, остава да я тестваме дали работи правилно. Извикваме уеб услугата от адрес:

http://localhost/AddService/AddService.asmx

Появява се следният прозорец, на който са изброени всички уеб методи от услугата:



В случая това е единствено методът **Add**. Щракаме с мишката върху него и се зарежда страница, в която можем са въведем стойности на входните параметри.



Въвеждаме две примерни числа, да кажем 2 и 3, и натиснем бутона **[Invoke]**. Това извиква уеб услугата. Резултатът се връща в XML формат:

```
<?xml version="1.0" encoding="utf-8" ?>
<int xmlns="http://tempuri.org/">5</int>
```

Ако въведем грешни входни данни, примерно текст вместо цяло число, се предизвиква изключение при опита да се конвертира текст до **Int32**:

```
Cannot convert alabala to System.Int32.
Parameter name: type --> Input string was not in a correct
format.
```

Използване на уеб услуги

Досега обяснихме как се създават уеб услугите. Тяхното основно предназначение обаче е да бъдат "консумирани", т.е. използвани от други

приложения. Това се осъществява посредством размяна на SOAP съобщения между отдалеченото приложение и уеб услугата.

За да се осъществи това обаче трябва да има механизъм, който да превръща заявките, отговорите и типовете от и към SOAP съобщения. В .NET Framework този механизъм се реализира чрез т. нар. междинни (proxy) класове. Междинните класове стоят между уеб услугата и отдалеченото приложение и вършат работата на преобразувател на SOAP съобщенията.

Междинните (proxy) класове

.NET Framework ни осигурява изключително лесен начин за създаване на междинни класове. Генерирането става автоматично от WSDL дефиницията на уеб услугата. За примера даден по-рано WSDL дефиницията е достъпна от адрес:

http://localhost/AddService/AddService.asmx?wsdl

Както всеки един клас в .NET Framework, така и междинните класове представляват сорс код на C# (Visual Basic .NET или някой друг език). В действителност ролята на междинните класове е да дадат на потребителя един лесен и типизиран начин за извикване, както синхронно така и асинхронно, на уеб методи от една услуга.

Генериране на междинен клас

Един лесен начин за генериране на междинни класове е чрез инструмента **wsdl**. За да генерираме междинен клас за последния пример е достатъчно да отворим Visual Studio .NET 2003 Command Prompt и да напишем:

```
wsdl http://localhost/AddService/AddService.asmx?wsdl
```

.NET Framework създава междинния клас за нашата уеб услуга и го поставя във файла **AddService.cs**. За всеки метод на уеб услугата в този клас се създават по 3 метода. Конкретно за последния пример **wsdl** генерира следните методи:

```
public AddService()
{
    this.Url = "http://localhost/AddService/AddService.asmx";
}

public int Add(int a, int b)
{
    object[] results = this.Invoke("Add", new object[] {
                                                    a,
                                                    b});
    return ((int)(results[0]));
}
```

```
public System.IAsyncResult BeginAdd(int a, int b,
    System.AsyncCallback callback, object asyncState)
{
    return this.BeginInvoke("Add", new object[] {
                                a,
                                b}, callback, asyncState);
}

public int EndAdd(System.IAsyncResult asyncResult)
{
    object[] results = this.EndInvoke(asyncResult);
    return ((int)(results[0]));
}
```

Add(…) се използва за синхронно извикване, а **BeginAdd(…)** и **EndAdd(…)** – за асинхронно извикване на уеб метода. Асинхронните извиквания на уеб услуги ще разгледаме по-късно в настоящата тема.

Използване на междинен клас

По своята същност прокси класовете не се различават от обикновените класове. Използват се по същия начин – първо се инстанцират (създава се обект от прокси класа) и след това се викат неговите методи:

```
AddService addSevice = new AddService();
int sum = addService.Add(5, 6);
```

Между извикването на обикновен клас и междинен клас, разбира се има няколко разлики.

Едната от тях е, че извикването може да се забави, тъй като трябва да се осъществи комуникация до отдалечен сървър.

При извикване на методите на междинния клас могат да се получат **SoapException** или **WebException**. **WebException** възниква при проблем с комуникацията с услугата (например ако сървърът е недостъпен), а **SoapException** възниква, ако е настъпил проблем на сървъра по време на изпълнението на уеб метода (например, ако е настъпило деление на 0).

По време на изпълнение на метод в уеб услугата изключенията, които могат да възникнат, могат да са от различен вид (InvalidCastException, DivideByZeroException, ...), но независимо от това при клиентското приложение пристига единствено SoapException, т.е. оригиналното изключение се губи.

В края на темата ще покажем описаният проблем със загубата на оригиналното изключение може да бъде заобиколен по прозрачен начин и при клиента да се получи оригиналното изключение.

Използване на междинен клас – пример

За да демонстрираме консумирането на уеб услуга чрез междинен клас, ще създадем клиентско приложение, което извиква създадената услуга **AddService**.

За целта на примера създаваме папка, в която ще държим нужните файлове. Нека за конкретност тя да е "**E:\Client**". Отваряме Visual Studio .NET 2003 Command Prompt и отиваме до папката "**E:\Client**", след което изпълняваме командата:

```
wsdl http://localhost/AddService/AddService.asmx?wsdl
```

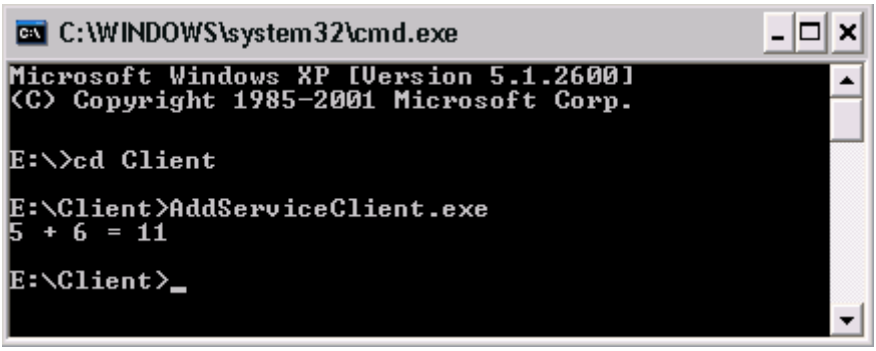
В резултат се създава междинният клас **AddService.cs**. В същата папка създаваме и файла "**AddServiceClient.cs**" със следното съдържание:

```
using System;

public class AddServiceClient
{
   static void Main()
   {
      AddService addService = new AddService();
      int a = 5;
      int b = 6;
      int sum = addService.Add(a, b);
      Console.WriteLine("{0} + {1} = {2}", a, b, sum);
   }
}
```

За да компилираме приложението, пишем в Command Prompt командата:

```
csc *.cs
```

В резултат получаваме компилираното асембли **AddServiceClient.exe**, което при изпълнение извежда следния резултат:

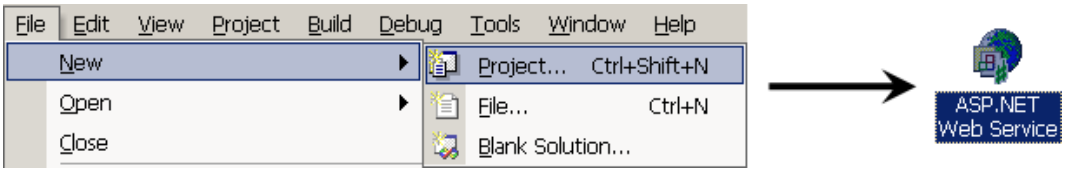


Уеб услугите и VS.NET – създаване и консумиране

Visual Studio .NET има силна поддръжка на уеб услуги. Създаването и консумирането им става почти автоматично – без да пишем допълнителен код на ръка.

Създаване на нова уеб услуга с VS .NET

За да създадем нова уеб услуга, отваряме Visual Studio .NET и от менюто избираме **File** | **New** | **Project**. От появилия се прозорец избираме ASP.NET Web Service.



В полето **Location** се появява адрес по подразбиране, на който да бъде създадена уеб услугата, примерно http://localhost/WebService1. Този адрес е върху локално инсталираният IIS.

Можем да променим този адрес на друг (примерно на http://localhost/NewWebService), стига на него да не съществува вече друго уеб приложение.

При създаване на нова уеб услуга Visual Studio .NET прави следното:

1. Създава `.sln` файл в директорията за проекти по подразбиране:

```
C:\Documents and Settings\<your user name>\My Documents\
Visual Studio Projects\<WebServiceName>\<WebServiceName>.sln
```

2. Създава виртуална директория в IIS в директорията на уеб сайта по подразбиране на IIS:

```
C:\Inetput\wwwroot\<WebAppName>
```

3. Създава в нея файловете на проекта:

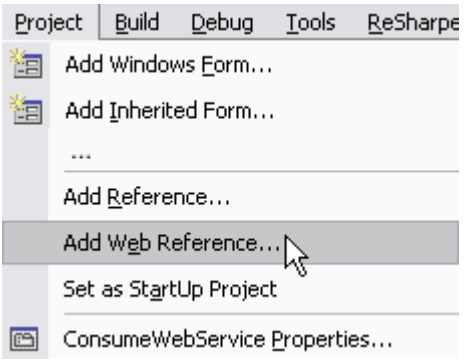
```
Bin\,
<WebServiceName>.csproj,
<WebServiceName>.csproj.webinfo,
<WebServiceName>.asmx,
<WebServiceName>.asmx.cs,
<WebServiceName>.asmx.resx,
AssemblyInfo.cs,
Web.config,
Global.asax,
Global.asax.cs,
Global.asax.resx
```

Отново се появява проблемът с правата, необходими за създаването на проекта в папката по подразбиране **C:\Inetput\wwwroot**. Освен това можем да искаме нашата уеб услуга да е разположена на друго място на твърдия диск.

За да решим тези проблеми най-лесно, преди да започнем създаването на уеб услугата можем да създадем папка там, където искаме да се създаде проекта, например в **E:\WebServices\NewService**, и по описания по-рано начин да настроим правата за достъп (Web Sharing и Security). След това трябва да регистрираме новата папка като виртуална директория в IIS и така при създаването на уеб услугата **NewWebService**, файловете й ще се намират в **E:\WebServices\NewService** вместо в **C:\Inetput\wwwroot**.

Консумиране уеб услуга с VS.NET

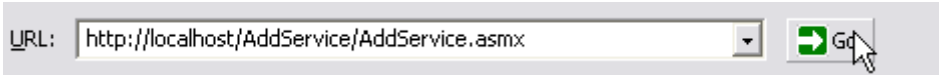
Ще демонстрираме как можем да използваме вече съществуваща уеб услуга от VS.NET. За целта нека създадем едно ново конзолно приложение **AddServiceConsole**. Искаме това приложение да използва услугата **AddService**. Добавяме връзка към уеб услугата като от менюто **Project** избираме **Add Web Reference**:



В резултат на това се появява прозорец, служещ за намиране на уеб услугата. Под полето **URL** се появяват няколко връзки, които позволяват лесно търсене на уеб услуги:

Първата връзка ни позволява да разгледаме уеб услугите, публикувани на локалната ни машина. Втората дава възможност да потърсим уеб услуга в локалната ни мрежа. Третата връзка ни праща в страницата за търсене на уеб услуги в UDDI бизнес регистъра. Четвъртата препратка ни води до страница за търсене на тестови уеб услуги, полезни при разработка и обучение.

Разбира се, винаги можем да си въведем директно адреса на уеб услугата в полето **URL** и да натиснем бутона **[Go]**, след което Visual Studio .NET започва да търси **wsdl** описанието на посочения адрес:



След като го намери, се активира полето **Web reference name**. В него записваме името на пространството, в което искаме да се генерират прокси класовете, примерно **MyService**. Натискаме след това бутона **[Add Reference]** и VS.NET създава за нас междинният (proxy) клас, за достъп до избраната уеб услуга.

За да използваме вече създаденият междинен клас и за да извикаме уеб услугата, поставяме следния код в **.cs** файла на приложението:

```
using System;

public class AddServiceClient
{
   using MyService;
   static void Main()
   {
      AddService addService = new AddService();
      int a = 5;
      int b = 6;
      int sum = addService.Add(a, b);
      Console.WriteLine("{0} + {1} = {2}", a, b, sum);
   }
}
```

С това създадохме същото приложение като това от Създаване на уеб услуги, само че с помощта на Visual Studio .NET.

Атрибути за уеб услугите

Уеб услугите използват два основни атрибута за описание. Това са **[WebService]** и **[WebMethod]**. Нека ги разгледаме по-внимателно.

Атрибутът [WebService]

Този атрибут се използва за описание на самия клас на услугата и се поставя точно преди неговата дефиниция. Чрез него могат да се задават

име, XML пространство от имена и кратко описание на услугата. Това става посредством полетата: **Name**, **Namespace** и **Description**.

Използването на XML пространства от имена позволява по уникален начин да се определят елементи или атрибути на XML документ. Описанието на една XML уеб услуга се дефинира в XML формат, и по специално чрез езика за описание на уеб услуги – WSDL.

В описанието на XML уеб услугите, полето **Namespace** от атрибута **WebService** се използва като пространство от имена по подразбиране за XML елементи, които директно принадлежат на уеб услугата. Например, името на уеб услугата и нейните методи принадлежат на пространството от имена, посочено в полето **Namespace**.

Добра практика е да промените стойността на полето **Namespace**, която е зададена подразбиране, тъй като тя идентифицира уникалността на уеб услугата. Ако всички уеб услуги, които осъществяват връзка помежду си, използват едно и също пространство от имена, ще се нарушат основни правила при описанието на SOAP пакетите и съответно тези уеб услуги няма да могат да работят заедно. Стойността на Namespace трябва да е уникален URI или URN идентификатор, както при всички XML пространства (вж. темата "Работа с XML").

Ето пример за използване на атрибута **[WebService]**:

```
[WebService(Namespace="http://localhost/xmlwebservices/",
     Description="Уеб услуга за събирането на две цели числа",
     Name="Add Service")]
public class AddService : System.Web.Services.WebService
{
   ...
}
```

Атрибутът [WebMethod]

Този атрибут указва, че даден метод е достъпен за клиентите на уеб услугата. Той има следните полета: **BufferResponse**, **CacheDuration**, **Description**, **EnableSession** и **MessageName**.

Полето BufferResponse

Това поле на атрибута **WebMethod** позволява буфериране на отговорите на уеб метода. Когато стойността му е **true**, ASP.NET буферира целия отговор преди да го изпрати към клиентското приложение. Буферирането е много ефикасно и спомага за подобряване на производителността като намалява комуникацията между работния процес и уеб сървъра. Ако стойността му е **false** ASP.NET буферира отговора на парчета от по 16KB. Стойността на това поле се слага на **false**, само когато не искаме да държим цялото съдържание на отговора в паметта наведнъж. Примерно такъв би бил случаят, ако връщаме съдържанието на колекция, която взема своите

елементи от база данни. Тогава задържането на целия отговор в паметта може да предизвика препълване на паметта. Ако не е указана стойност за това поле, стойността по подразбиране е **true**.

Полето CacheDuration

Това поле отговаря за кеширането на резултатите от даден уеб метод. Кеширането се използва за да увеличи производителността на сървъра, когато даден уеб метод променя рядко връщания резултат.

ASP.NET кешира резултатите за всяко множество входни параметри, т. е. при подадени различни параметри на уеб метода няма да се върне кешираният резултат, а ще се изпълни методът с новите параметри и резултатът също ще се кешира. При повторно извикване на метода с едни и същи параметри в рамките на зададения период се връща вече кешираният резултат. Стойността на това поле определя за колко секунди ASP.NET да кешира резултата. Стойността 0 означава, че кеширането е изключено. Ако не е указана, стойността за това поле, по подразбиране е 0.

Полето Description

Чрез това поле се задава кратко описание за уеб метода, което се появява в help страницата на услугата. Ако не е указана, стойността за това поле, по подразбиране се използва празен символен низ.

Полето EnableSession

Стойността на това поле определя дали за дадения уеб метод е позволено използването на сесия. Ако тя е **true**, уеб услугата има достъп до сесията чрез **HttpContext.Current.Session** или чрез полето **WebService.Session**, ако се наследява базовият клас **WebService**. Ако не е указана, стойността за това поле по подразбиране е **false**. Работата със сесии използва Cookies и това може да предизвика несъвместимости с други платформи, тъй като Cookies не са част от спецификациите за уеб услуги, а са възможност на HTTP протокола. По-нататък в настоящата тема ще разгледаме в детайли проблема за поддръжка на сесии.

Полето MessageName

Задавайки стойност на това поле на атрибута **WebMethod**, можем да сменим името на метода при клиента. Това позволява на услугата да постави уникални имена на препокриващи се методи. Ако не е указана стойност за това поле, стойността по подразбиране е самото име на метода. Когато е зададена стойност на полето **MessageName**, резултатното SOAP съобщение от метода ще отговаря на зададеното име вместо на името на истинския метод в класа на уеб услугата.

Прехвърляне на типове (marshalling)

Прехвърлянето на типове (маршализация) е процесът на трансформация на различните типове данни от SOAP и XML към .NET типове и обратното.

На всеки .NET тип се съпоставя съответен SOAP тип и обратното – на всеки SOAP тип, описан в WSDL дефиницията на услугата, се съпоставя .NET тип. Прехвърлянето на типовете има и своите особености, с които трябва да се съобразяваме.

Не всички .NET типове могат да се прехвърлят през уеб услуга. Например няма как да прехвърлим на отдалечена машина отворен файл. Някои по-сложни типове, например рекурсивните структури от данни, също не могат да се прехвърлят директно, защото SOAP и WSDL стандартите са по-общи и не са съобразени с всички особености на .NET типовете.

Различни типове данни могат да се предават като параметри на уеб метод и да се връщат като резултат. Когато някакъв тип данни, обект или метод се подаде като SOAP заявка или отговор, той автоматично се прехвърля във вид на XML.

Тъй като всеки език за програмиране може да използва SOAP, SOAP дефинира свои собствени типове данни. Когато се подадат някакви данни в SOAP съобщение, те се прехвърлят в техен SOAP еквивалент. Това позволява на различните езици с различни имена на типовете да комуникират ефективно. Фактът, че уеб услугите са базирани на XML сериализация, позволява значителен брой типове данни да бъдат прехвърляни (вж. темата "Сериализация на данни").

Примитивни типове

Стандартните примитивни типове се прехвърлят директно и безпроблемно. Това са типовете: **string**, **char**, **byte**, **bool**, **sbyte**, **int**, **uint**, **long**, **ulong**, **short**, **ushort**, **float**, **double**, **decimal**. Без проблеми се прехвърлят и някои стандартни структури, като **Guid** и **DateTime**. Например символният низ "Hello World" се прехвърля във вида:

```
<string>Hello World</string>
```

Числовите типове също се прехвърлят в текстов вид в строго определен от XML Schema (XSD) спецификацията формат.

Изброени типове

Всички изброени типове се прехвърлят под формата на низове като се вземат имената на изброените им стойности: **enum Color {Red, Blue}**.

Класове и структури

От класовете и структурите се само публичните полета и свойства. Поддържат се вложени типове и дървовидни структури, но не и циклични типове. Задължително условие за да може да се прехвърли един клас е той да има дефиниран конструктор без параметри. Ето няколко примера за структури и класове, които се прехвърлят безпроблемно:

```
public struct Point
```

```
{
   public int x, y;
}

public class Student
{
   public int Age
   {
      get {…}
      set {…}
   }
}
```

Тези типове се прехвърлят във вид на XML по следния начин:

```
<Point>
   <x xsi:type="xsd:int">5</x>
   <y xsi:type="xsd:int">5</y>
</Point>

<Student xsi:type="ns1:Student" >
   <Age xsi:type="xsd:int">13</Age>
</Student>
```

Въпреки, че горните методи не дефинират изрично конструктор без параметри, той се дефинира по подразбиране от компилатора, тъй като тези типове не дефинират нито един друг конструктор (вж. темата "Обектно-ориентирано програмиране в .NET").

Масиви

Масивите от примитивни типове, изброени типове, класове или структури също се прехвърлят без проблеми: **`string[]`**, **`Color[]`**, **`Point[]`**. За пример можем да разгледаме следният масив от символни низове:

```
string[] emailAddresses = new string[] {
   "testEmail1@testDomain.com",
   "testEmail2@testDomain.com"};
```

Той се прехвърля при пренасяне в SOAP съобщения в следния формат:

```
<emailAddresses xsi:type="SOAP-ENC:Array"
  SOAP-ENC:arrayType="xsd:string[2]">
   <item xsi:type="xsd:string">
      testEmail1@testDomain.com
   </item>
   <item xsi:type="xsd:string">
      testEmail2@testDomain.com
   </item>
</emailAddresses>
```

Колекции

Колекциите от примитивни типове, изброени типове, класове или структури се прехвърлят като масиви. Това означава, че ако даден уеб метод връща колекция (например **ArrayList**), в WSDL описанието вместо колекция типът ще бъде дефиниран като масив. В резултат на това и в междинния (proxy) клас методът ще връща масив. За пример можем да разгледаме извикването на следния уеб метод:

```
[WebMethod]
public ArrayList HelloWorld()
{
   ArrayList list = new ArrayList();
   list.Add("item 1");
   list.Add("item 2");
   list.Add(42);
   return list;
}
```

При извикването на този метод резултатът се прехвърля във вид на XML в следния формат:

```
<ArrayOfAnyType xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://localhost/xmlwebservices/">
   <anyType xsi:type="xsd:string">item 1</anyType>
   <anyType xsi:type="xsd:string">item 2</anyType>
   <anyType xsi:type="xsd:int">42</anyType>
</ArrayOfAnyType>
```

DataSet обекти

DataSet обектите също могат да се предават през уеб услуги. Тяхното представяне е доста обемисто и затова трябва да се внимава, тъй като те значително увеличават обема на информация, прехвърляна между услугата и приложенията. При прехвърляне на **DataSet** обект се прехвърля XSD схемата му, последвана от сериализираните данни за всяка от таблиците му под формата на XML.

XmlNode

Всякакви XML фрагменти могат да се предават между клиента и услугата в чист вид. В .NET Framework тези XML фрагменти съответстват на класа **System.Xml.XmlNode**.

Прехвърляне на типовете – пример

За да демонстрираме прехвърлянето на типове между клиент и уеб услуга, ще създадем примерна уеб услуга **TypesService** с няколко уеб метода. По описания по-рано начин, създаваме първо услугата, а след това и помощен клас, структура и енумерация:

```
public enum Color
{
   Red,
   Blue
};

public struct Point
{
   public int x, y;
}

public class Student
{
   // Private fields - not serialized and marshalled
   private string mName;
   private int mAge;
   private int mState = 0;

   public string Name
   {
      get
      {
         return mName;
      }
      set
      {
         mName = value;
      }
   }

   public int Age
   {
      get
      {
         return mAge;
      }
      set
      {
         mAge = value;
      }
   }

   // This parameterless public constructor
   // is required for the XML serialization
   public Student()
   {
   }

   public Student(string aName, int aAge)
   {
```

```
      mName = aName;
      mAge = aAge;
   }
}
```

Добавяме следните уеб методи:

`GetColors()` – връща масив от тип **`Color`**. Ще върне елементите на изброения тип **`Color`**:

```
[WebMethod(Description="Returns a list of available colors.")]
public Color[] GetColors()
{
   Color[] colors = new Color[2];
   colors[0] = Color.Blue;
   colors[1] = Color.Red;

   return colors;
}
```

`CalculateDistance(…)` – приема като параметър две променливи от тип **`Point`** и връща като резултат разстоянието между двете точки в Евклидово пространство:

```
[WebMethod(Description="Calculates distance between two points
in the plane.")]
public double CalculateDistance(Point p1, Point p2)
{
   int dx = p1.x - p2.x;
   int dy = p1.y - p2.y;
   double distance = Math.Sqrt(dx*dx + dy*dy);
   return distance;
}
```

`ConvertDegreesToRadians(…)` – приема променлива от тип **`double`**, представляваща големината на ъгъл в градуси и връща стойността му в радиани в същата променлива. На практика променливата е входно-изходна. Такива променливи се поддържат стандартно от уеб услугите:

```
 [WebMethod(Description="Converts given angle from degrees to
radians.")]
public void ConvertDegreesToRadians(ref double aAngle)
{
   aAngle = (double) aAngle * Math.PI / 180;
}
```

`GetStudents()` – връща примерен масив от обекти от тип **`Student`**:

```
[WebMethod(Description="Returns a list of Student objects.")]
```

```
public Student[] GetStudents()
{
   Student[] students = new Student[3];
   students[0] = new Student("Иван", 20);
   students[1] = new Student("Мария", 19);
   students[2] = new Student("Жоро", 21);
   return students;
}
```

GetDataSet() – създава примерен **DataSet** и го връща като резултат:

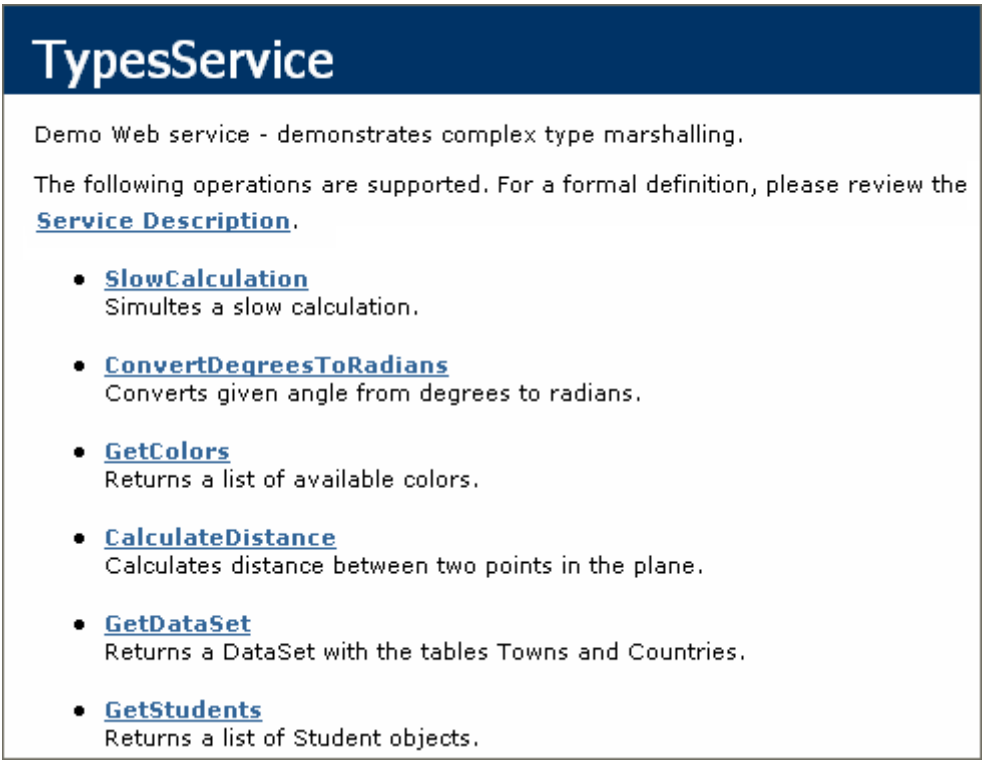
```
[WebMethod(Description="Returns a DataSet with the tables Towns
and Countries.")]
public DataSet GetDataSet()
{
   DataTable towns = new DataTable("Towns");
   towns.Columns.Add("id", typeof(int));
   towns.Columns.Add("name", typeof(string));

   DataRow sofia = towns.NewRow();
   sofia["id"] = 1;
   sofia["name"] = "София";
   towns.Rows.Add(sofia);

   DataRow varna = towns.NewRow();
   varna["id"] = 2;
   varna["name"] = "Варна";
   towns.Rows.Add(varna);

   DataTable countries = new DataTable("Countries");
   countries.Columns.Add("id", typeof(int));
   countries.Columns.Add("name", typeof(string));

   DataRow bg = countries.NewRow();
   bg["id"] = 1;
   bg["name"] = "България";
   countries.Rows.Add(bg);

   DataSet ds = new DataSet();
   ds.Tables.Add(towns);
   ds.Tables.Add(countries);
   return ds;
}
```

SlowCalculation() – приспива нишката на изпълнение на метода за да симулира времеотнемащо изпълнение. Ще използваме този метод по-нататък, когато разглеждаме асинхронните извиквания:

```
[WebMethod(Description="Simultes a slow calculation.")]
public int SlowCalculation()
```

```
{
   System.Threading.Thread.Sleep(3000);
   return 0;
}
```

Ако компилираме сега уеб услугата и я заредим през браузъра ще видим списъка от методи, които са на разположение да бъдат извикани и краткото описание зададено в полето **`Description`** на атрибута **`WebMethod`** за всеки от тях:



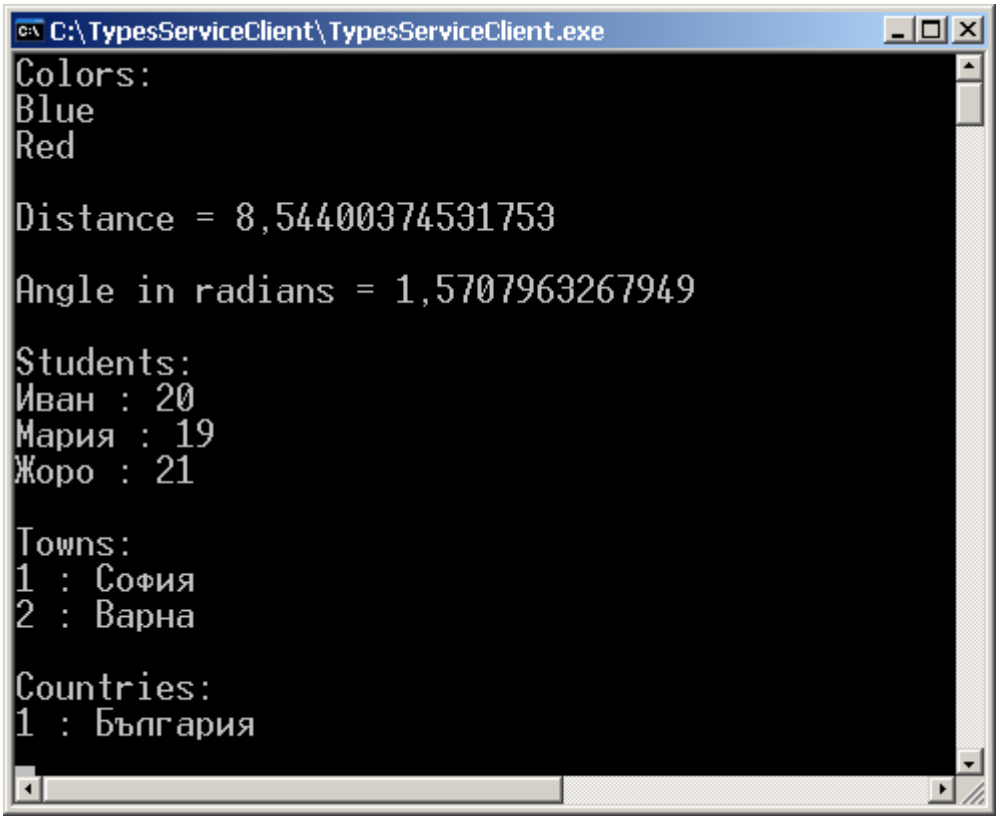
Можем да извикваме методите, за да тестваме услугата през уеб браузъра или можем да използваме специално написан за услугата клиент:

```
static void Main()
{
   MyServices.TypesService service =
      new MyServices.TypesService();

   // Invoke GetColors() Web method
   MyServices.Color[] colors = service.GetColors();
   Console.WriteLine("Colors:");
   foreach (MyServices.Color color in colors)
   {
      Console.WriteLine(color);
   }
```

```
  // Invoke CalculateDistance() Web method
  MyServices.Point p1 = new MyServices.Point();
  p1.x = 4;
  p1.y = 5;
  MyServices.Point p2 = new MyServices.Point();
  p2.x = 7;
  p2.y = -3;
  double distance = service.CalculateDistance(p1,p2);
  Console.WriteLine("\nDistance = {0}", distance);

  // Invoke ConvertDegreesToRadians(double angle)
  double angle = 90;
  service.ConvertDegreesToRadians(ref angle);
  Console.WriteLine("\nAngle in radians = {0}", angle);

  // Invoke GetStudents()
  MyServices.Student[] students = service.GetStudents();
  Console.WriteLine("\nStudents:");
  foreach (MyServices.Student student in students)
  {
    Console.WriteLine("{0} : {1}", student.Name, student.Age);
  }

  // Invoke GetDataSet()
  DataSet ds = service.GetDataSet();

  DataTable towns = ds.Tables["Towns"];
  Console.WriteLine("\nTowns:");
  foreach (DataRow town in towns.Rows)
  {
    Console.WriteLine("{0} : {1}", town["id"], town["name"]);
  }

  DataTable countries = ds.Tables["Countries"];
  Console.WriteLine("\nCountries:");
  foreach (DataRow country in countries.Rows)
  {
    Console.WriteLine("{0} : {1}",
      country["id"], country["name"]);
  }
}
```

Горният клиент разчита на типовете от пространството **`MyServices`**, които се генерират автоматично от VS.NET по WSDL описанието на услугата. Тези типове съответстват на оригиналните .NET типове, дефинирани в услугата, но реално са локални класове, структури и енумерации, дефинирани в междинните (proxy) класове на клиента.

Ето и как изглежда резултатът от изпълнението на горния примерен клиентски код:



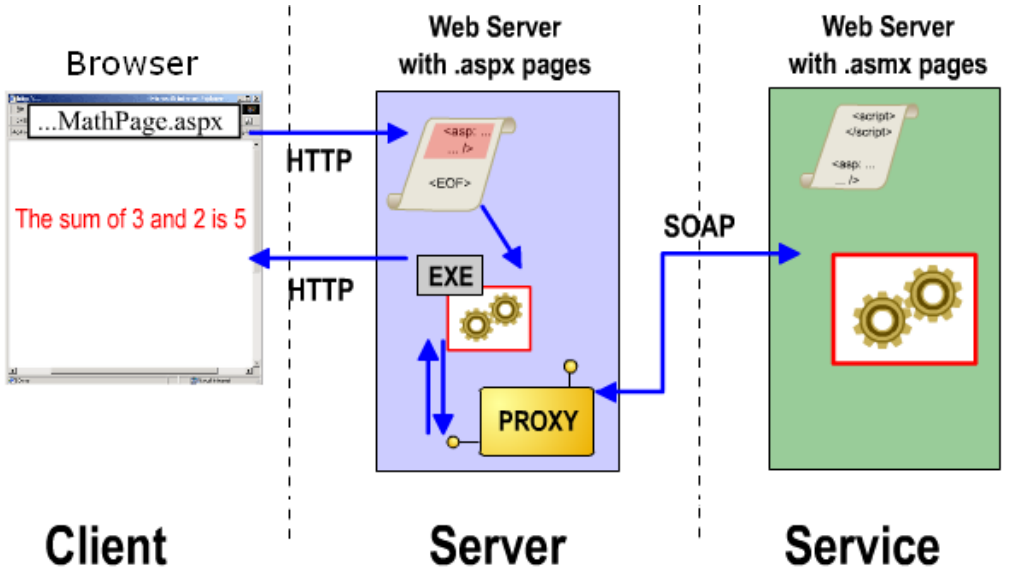
Дебъгване на уеб услуги

По отношение на дебъгването уеб услугите по нищо не се различават от уеб приложенията. Те се дебъгват по абсолютно аналогичен начин. Visual Studio .NET ни дава възможността както да стартираме направо услугата в режим Debug, така и да се прикачим към работния процес, обслужващ ASP.NET. Тези двата подхода за дебъгване са подробно описани в темата "Уеб приложения с ASP.NET".

Моделът на изпълнение на уеб услугите в ASP.NET

Уеб услугите са създадени за да бъдат използвани от други приложения. Те нямат графичен потребителски интерфейс и когато потребителят иска да се възползва от някоя уеб услуга, той трябва да се обърне към някое приложение, което я използва.

Често пъти клиенти на уеб услугите са ASP.NET уеб приложения. Моделът на изпълнение при такива системи можем да представим схематично по следния начин:



1. Клиентът изпраща заявка към уеб сървъра за определена **`.aspx`** страница.
2. Уеб сървърът извиква ISAPI библиотеката на .NET Framework – **`aspnet_isapi.dll`**, която се грижи за понататъшната обработка на заявката. .NET Framework парсва **`.aspx`** страницата, компилира я и я изпълнява (Това става в случай, че заявка към тази страница се подава за първи път. Ако не е така, направо се изпълнява вече зареденият код).
3. По време на изпълнението на компилирания код, ASP.NET приложението се обръща към методи на междинния (proxy) клас, който е генериран и компилиран заедно с приложението. Тези методи приемат същите параметри и връщат същия резултат както и реализираните в уеб услугата.
4. Междинните (proxy) методи от своя страна конструират SOAP съобщения и ги изпращат до уеб услугата. Това става по същия начин, по който се извикват **`.aspx`** страници – отново чрез HTTP заявка към уеб сървъра и извикване на ISAPI филтъра за ASP.NET.
5. Уеб услугата има свой собствен модел на вътрешно изпълнение. SOAP съобщението се парсва и се извиква съответният метод. В резултат се връща някакъв резултат, който отново се сериализира в SOAP формат и се връща на уеб приложението.
6. Междинният (proxy) метод десериализира SOAP съобщението и връща резултата като .NET обект.
7. ASP.NET уеб приложението довършва изпълнението си и връща отговор на заявката на клиента под формата на HTML страница.

В стъпка номер 5 от изпълнението на показания модел споменахме за вътрешния модел на изпълнение за уеб услугата. Той описва процеса, който се изпълнява при извикване на уеб услуга от момента на постъпване на SOAP заявката до момента, в който се връща отговор. За всяка заявка към уеб услуга ASP.NET изпълнява следното:

1. Инстанцира се класът на уеб услугата.
2. Заделя се отделна нишка от общия пул с нишки.
3. Изпълнява се заявката в тази нишка – извиква определения метод.
4. Връща се резултата към клиента – отново под формата на SOAP съобщение.
5. Нишката се връща обратно в общия пул с нишки.
6. Оставя инстанцията на класа на услугата да бъде унищожена от системата за почистване на паметта (garbage collector).

Естеството на уеб услугите се състои в това те да могат да бъдат използвани от много приложения едновременно, което води до необходимостта много заявки да се обслужват едновременно. Нишките за изпълнение, както всички останали ресурси в реалния свят, са краен брой. При достатъчно голяма натовареност на уеб услугата, заявките стават повече от наличните нишки. В такъв случай ASP.NET поставя заявките в опашка и когато се освободи нишка, в нея започва изпълнението на следващата заявка от опашката.

Асинхронно извикване на уеб услуги

Когато създаваме приложения, искаме те винаги да отговарят на потребителското взаимодействие, независимо, че може приложението да извършва някоя тежка и времеотнемаща операция. Ако една такава операция съдържа в себе си извикване на уеб услуга (особено, ако това става през Интернет, а не на локалната машина), изпълнението може да отнеме голямо количество време.

Практиката показва, че забавяния от порядъка на 200 милисекунди до цяла секунда, са често срещани. Това може и да не изглежда много, но за потребителите това е сериозно дразнение, понеже потребителският интерфейс напълно спира да функционира за този период.

Потребителският интерфейс обикновено блокира докато се извиква уеб метод, защото той, заедно с уеб метода работят в една и съща нишка. Докато уеб методът не върне резултат, нишката блокира изпълнението си и така потребителският интерфейс не се обновява.

Изпълнението на всеки метод, който предизвиква значително забавяне, в основната нишка предизвиква блокиране на потребителския интерфейс. Това трябва да се избягва. Такова поведение на приложението може да накара потребителя да започне да хвърля клавиатури, да троши монитори, или още по-лошо – никога повече да не купува вашия софтуер.

Методи за асинхронно извикване в междинния (proxy) клас

Когато създаваме междинен (proxy) клас за дадена уеб услуга, .NET Framework генерира в него методи, напълно аналогични на тези в уеб услугата. Ако отворим сорс кода освен стандартните методи обаче ще забележим и такива, чиито имена започват с "**Begin**" и "**End**" и завършват с името на уеб метод. Например ако уеб услугата има уеб метод с име **CalculatePayment()**, в междинния (proxy) клас ще има съответно метод **CalculatePayment()**, а също така и два метода **BeginCalculatePayment()** и **EndCalculatePayment()**. Последните се използват при асинхронното извикване на уеб услуги.

Тъй като при асинхронното извикване се заделя допълнителна нишка, независима от основната, е необходимо да дефинираме метод, който да бъде извикан обратно при приключване на работата на допълнителната нишка. Този метод задължително трябва да приема параметър от тип **IAsyncResult**.

За да демонстрираме асинхронно извикване на уеб метод, ще използваме дефинирания в предната демонстрация уеб метод - **SlowCalculation()**. от услугата **TypesService**.

За целта създаваме ново конзолно приложение с име **AsynWSCallDemo** и в него добавяме файла **AsynWSCallDemo.cs** и уеб референция към създадената в предната демонстрация уеб услуга (**TypesService**). Ето сорс кода на примерното приложение:

AsynWSCallDemo.cs

```
using System;
using AsynWSCallDemo.MyServices;

namespace AsynWSCallDemo
{
   class AsynWSCallDemo
   {
      private static TypesService mService = new TypesService();

      public static void Main()
      {
         AsyncCallback cb = new AsyncCallback(CallFinished);
         IAsyncResult ar =
            mService.BeginSlowCalculation(cb, mService);
         Console.WriteLine("Async call started.");
         Console.Write("Loading.");
         int cycleCounter = 0;
         while(!ar.IsCompleted)
         {
            cycleCounter++;
         }
```

```
        Console.WriteLine("Cycles Passed: " + cycleCounter);

        Console.ReadLine();
      }

      private static void CallFinished(IAsyncResult aAsyncResult)
      {
        Console.WriteLine("Async call completed.");
        int result = mService.EndSlowCalculation(aAsyncResult);
        Console.WriteLine("Result = {0}", result);
      }
    }
}
```

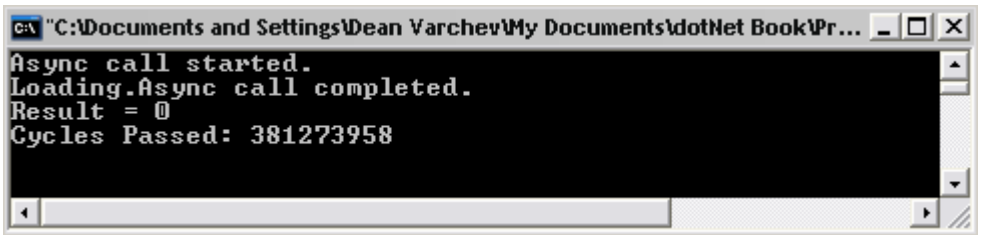
В основния метод **Main(…)** създаваме обект от тип **AsyncCallback**, на който в конструктора подаваме метода, който да бъде извикан при приключване на работата на допълнителната нишка. Този обект трябва да се създаде, за да бъде подаден след това като параметър на метода, извикващ асинхронно уеб метода на уеб услугата.

Инстанцията на уеб услугата в примера е **mService**. Извикваме метода на **mService BeginSlowCalculation(…)** като му подаваме множество параметри. В случая уеб методът на услугата няма входни параметри затова подаваме задължителните **cb** (обекта от тип **AsyncCallback**) и **mService**.

Ако на уеб метода се подават някакви параметри, тогава на **BeginSlowCalculation(…)** първо се подават те и след това **AsyncCallback** и **WebService** обектите.

Като резултат **BeginSlowCalculation(…)** връща обект от тип **IAsyncResult**, който може да използваме за проследяване на състоянието на изпълнение на уеб метода. В случая само увеличаваме стойността на брояч, като след приключване на изпълнението на уеб метода извеждаме стойността му на екрана. Чрез полето **IsCompleted** на обекта, върнат от извикването на метода **BeginSlowCalculation(…)**, проверяваме дали е приключило асинхронното изпълнение на уеб метода.

Методът **CallFinished(…)** приема аргумент също от тип **IAsyncResult**, като този обект се използва при извикването на втория метод **EndSlowCalculation(…)**, с който получаваме резултата, върнат от асинхронното извикване. Резултатът от изпълнението на горния пример е следният:



Уеб услуги и работа с данни

Много често уеб услугите се ползват като междинен слой в трислойните приложения. В един такъв сценарий тяхната основна задача е извличането и обработката на данни от базата данни, както и приемането на данни и вкарването им обратно в базата. Чрез тях най-често се реализира основната бизнес логика на приложенията.

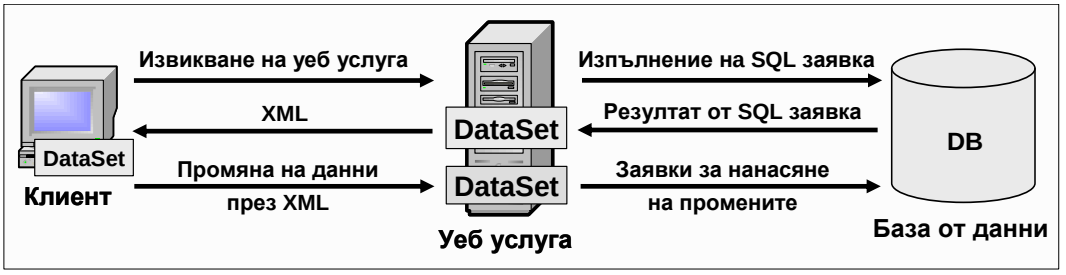
Възможността на уеб услугите да приемат и връщат почти всякакъв тип данни, предоставя изключително голямо разнообразие от области на приложение. В зависимост от приложенията консуматори, най-често се прехвърлят `DataSet` обекти или масиви от данни. Използването на `DataSet`, обаче, е свързано най-вече с .NET приложенията и рядко се ползва при хетерогенни системи.

При приложения, които не поддържат обекти от тип `DataSet`, за пренос на данни се използват така наречените обекти за пренос на данни (DTO – Data Transfer Objects). При преноса на данни, тези обекти се сериализират под формата на XML, а самият клас на обекта се дефинира в WSDL описанието на услугата. В действителност се сериализират всички публични полета и свойства на обекта.

Когато Visual Studio .NET създава междинен (proxy) клас за услугата, той генерира и клас от типа на обектите за пренос на данни, който е без методи, а се състои само от публични полета. По същия начин, ако услугата се използва не от.NET, а от Java, Perl, PHP или от друга платформа, съответните DTO обекти се преобразуват от XML към обекти от съответния език за програмиране и платформа.

Уеб услугите и работа с данни в .NET

Типичният модел на работа с данни при .NET уеб услуга и .NET клиент е илюстриран схематично на следната фигура:



Типичният сценарий включва следните стъпки:

1. Клиентското приложение извиква метод от уеб услугата.
2. В уеб услугата се изпълняват поредица от логически операции и евентуално SQL заявка към базата от данни.
3. Резултатът от SQL заявката се връща на уеб услугата, където той се оформя във вид на `DataSet` обект или обект за пренос на данни.

4. Този обект се сериализира в XML и в такава форма се връща на клиента.
5. Клиентското приложение извършва промяна на данните и отново под формата на XML ги връща обратно на уеб услугата за запис.
6. Услугата, от своя страна, изпълнява друга заявка към базата от данни, с която записва променените данни.

Уеб услугите и работа с данни в .NET – пример

За да демонстрираме работата с данни в уеб услугите, ще създадем проста уеб услуга, съдържаща само два уеб метода. Единият ще служи за извличане на данните от базата, а другия за тяхната промяна. След това ще създадем и клиент за уеб услугата, който чете данните, прави промени по тях и връща промените обратно в уеб услугата

Уеб услуга за работа с данни в .NET

При изграждането уеб услугата ще използваме един **`SqlDataAdapter`** за достъп и промяна на данните от таблицата "**`Categories`**" на базата данни "**`Northwind`**" в MS SQL Server.

Чрез Drag and Drop на таблицата "**`Categories`**" създаваме **`SqlDataAdapter`** и го именуваме **`sqlDataAdapterCategories`**. Visual Studio .NET автоматично генерира за нас всички команди, които са нужни за достъп до съответната таблица в базата данни. От контекстното меню на **`sqlDataAdapterCategories`** създаваме **`DataSet`** клас **`DataSetCategories`**.

Вече сме готови с **`DataSet`** обекта и адаптера за таблицата **`Categories`**. Създаваме и два уеб метода **`GetCategories()`** и **`UpdateCategories(…)`**, като вторият метод приема като параметър **`DataSet`**, съдържащ промените, нанесени от клиента:

```
[WebMethod(Description="Gets all category entries as DataSet.")]
public DataSet GetCategories()
{
   DataSetCategories dsCategories = new DataSetCategories();
   sqlDataAdapterCategories.Fill(dsCategories);
   return dsCategories;
}

[WebMethod(Description="Updates the category entries by given
change list.")]
public void UpdateCategories(DataSet aDatasetCategoriesChanges)
{
   sqlDataAdapterCategories.Update(aDatasetCategoriesChanges);
}
```

Така уеб услугата вече е готова и можем да я стартираме чрез натискане на [**`F5`**] и да извикаме метода **`GetCategories()`**. Като резултат той връща

сериализиран **DataSet** обект, съдържащ таблицата **Categories**, като заедно с него идва и описваща го XSD схема.

Клиент за работа с данни в .NET

Нека сега създадем клиентско Windows Forms приложение, което да извиква методите на уеб услугата. За целта създаваме нов Windows Forms проект, към който добавяме уеб референция към създадената уеб услуга (http://localhost/NorthwindService/NorthwindService.asmx). Преименуваме генерираната форма на **MainForm**. Като частно поле на формата прибавяме инстанция на уеб услугата:

```
private MyServices.NorthwindService mNorthwindService =
  new MyServices.NorthwindService();
```

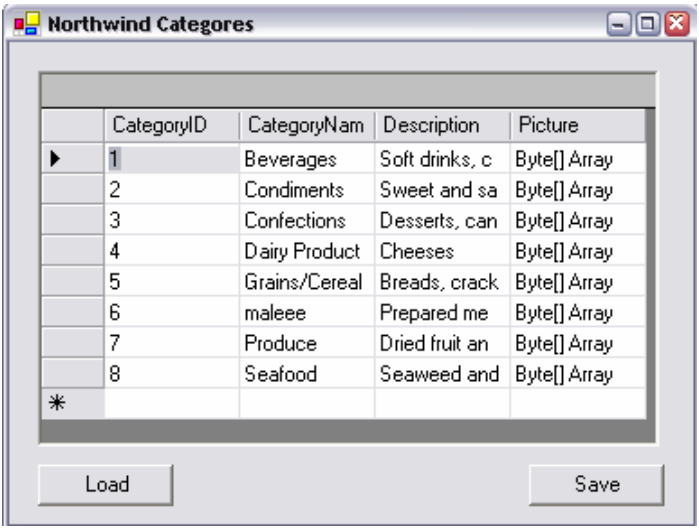
Във формата добавяме бутона за зареждане на данните, бутон за записване на данните и **DataGrid** контрола. Кръщаваме ги съответно **buttonLoad**, **buttonSave** и **dataGridCategories**. Към събитието **Click** на двата бутона прикачваме съответно методите **buttonLoad_Click** и **buttonSave_Click**, които извикват съответно **LoadData()** и **SaveData()**. При събитието **Load** на **MainForm** извикваме също **LoadData()**. Методите **LoadData()** и **SaveData()** изпълняват следния код:

```
private void LoadData()
{
  try
  {
    // Load the categories table from the Web service
    mDsCategories = mNorthwindService.GetCategories();

    // Bind the data to the DataGrid control
    dataGridCategories.DataSource = mDsCategories;
    dataGridCategories.DataMember = "Categories";
  }
  catch (Exception)
  {
    MessageBox.Show(
      "Can not retrive the categories from the server.",
      "Error");
  }
}

private void SaveData()
{
  try
  {
    DataSet dsCategoriesChanges = mDsCategories.GetChanges();
    if (dsCategoriesChanges != null)
    {
```

```
        // Commit the changes to the Web service
        mNorthwindService.UpdateCategories(dsCategoriesChanges);

        MessageBox.Show("Categories updated succesfully.",
           "Info");
      }
   }
   catch (Exception)
   {
      MessageBox.Show(
         "Can not update the categories.", "Error");
   }

   // Refresh the categories table
   LoadData();
}
```

Методът **LoadData()** извиква метода на уеб услугата **GetCategories()**, който връща **DataSet** обект с таблицата "**Categories**". Резултатът се присвоява на **DataSource** свойството на **dataGridCategories** и на свойството **DataMember** се присвоява символен низ с името на таблицата – "**Categories**". Така се извършва свързване на таблицата с **DataGrid** контролата (data binding). Целият този код е заграден в **try-catch** блок, в който се прихващат евентуално възникналите изключения.

Методът **SaveData()** първо взема **DataSet**, който съдържа променените данни и с този **DataSet** извиква метода **UpdateCategories(…)** на уеб услугата и накрая вика отново **LoadData()**. На услугата се предават само направените промени (ако има). След това се извлича цялата таблица наново, за да се работи с актуални данни. Това зарежда промените, които други потребители междувременно са нанесли в базата данни.



Поддръжка на сесии

По отношение поддръжка на сесии уеб услугите са абсолютно аналогични на уеб приложенията. Сесиите при уеб услугите също представляват инстанции на класа **HttpSessionState** както и в уеб приложенията. Същи са и обектите за достъп до сесии – **Session** за достъп до текущата сесия и **Application** за достъп до контекст на цялото приложение.

Точно както в уеб приложенията и уеб услугите могат да бъдат настройвани от уеб конфигурационния файл. Настройките включват дали да се използват Cookies, къде да се държи сесията – в паметта, в SQL Server, или на друго място, след колко време да изтече сесийното Cookie и т. н.

Ако искаме да не използваме cookie, задаваме **true** в атрибута **cookieless** на тага **sessionState**. Трябва да се има предвид, че при използването на cookieless сесия се получават известни проблеми в клиентските приложения. Малко по-късно ще се спрем на тях, когато описваме консумацията на уеб услуги, използващи сесии.

За да се разреши на даден уеб метод да използва сесията, е необходимо единствено да се зададе на полето **EnableSession** на атрибута **WebMethod** стойност **true**. По подразбиране всички уеб методи не поддържат сесия.

Сесии в уеб услуги – пример

За да демонстрираме как се използват сесии в уеб услугите, ще създадем примерна уеб услуга, която да използва обектите **Session** и **Application**. Тя ще се състои от два уеб метода – **GetSessionCounter()** и **GetApplicationCounter()**:

```
[WebMethod(EnableSession=true, Description="Returns the next
value of the local session counter.")]
public int GetSessionCounter()
{
   int counter = 0;
   if (Session["counter"] != null)
   {
      counter = (int) Session["counter"];
   }
   counter++;
   Session["counter"] = counter;
   return counter;
}

[WebMethod(Description="Returns the next value of the global
application counter.")]
public int GetApplicationCounter()
{
   Application.Lock();
   try
```

```
    {
        int counter = 0;
        if (Application["counter"] != null)
        {
            counter = (int) Application["counter"];
        }
        counter++;
        Application["counter"] = counter;
        return counter;
    }
    finally
    {
        Application.UnLock();
    }
}
```

Методът **GetSessionCounter()** инициализира един брояч при първо извикване, а ако той вече се съдържа в локалната сесия, неговата стойност се взема от там. След това стойността на брояча се увеличава с единица, отново се поставя в сесията и се връща като резултат от извикването на метода. Целта е да се демонстрира, че в сесията може да се държи информация, която се запазва между отделните извиквания.

Тъй като **Application** обектът е общ за всички инстанции на уеб услугата, чиито методи се изпълняват понякога едновременно, е необходимо достъпът до него да е синхронизиран. Методът **GetApplicationCounter()** първо извършва точно това – забранява достъпа до обекта от други инстанции на услугата и след това прави абсолютно същото както и **GetSessionCounter()**, като накрая отново разрешава достъпа към **Application** обекта.

Сесии в уеб услуги – клиентско приложение

В зависимост от вида на сесията на уеб услугата се разрешава и използването на сесии в клиентско приложение. Ако сесията на услугата използва Cookies, единственото, което трябва да се направи при клиента, е да се добави **System.Net.CookieContainer** към уеб услугата:

```
mSessionService = new SessionService();
mSessionService.CookieContainer =
    new System.Net.CookieContainer();
```

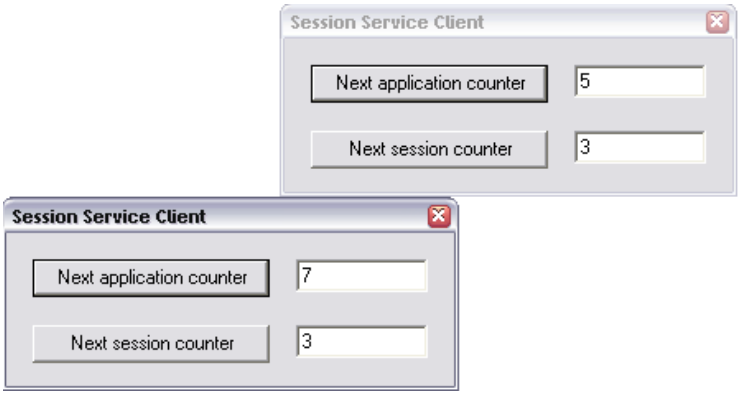
Нека създадем просто Windows Forms приложение, чиято форма се състои само от две текстови полета – **textBoxNextSessionCounter** и **textBoxNextAppCounter** и два бутона – **buttonNextSessionCounter** и **buttonNextAppCounter**. При събитието **Click** на бутоните се изпълняват съответно методите:

```
private void buttonNextSessionCounter_Click(object sender,
```

```
   System.EventArgs e)
{
   int nextValue = mSessionService.GetSessionCounter();
   textBoxNextSessionCounter.Text = nextValue.ToString();
}

private void buttonNextAppCounter_Click(object sender,
   System.EventArgs e)
{
   int nextValue = mSessionService.GetApplicationCounter();
   textBoxNextAppCounter.Text = nextValue.ToString();
}
```

Обработчиците за натискане на бутон извикват създадените в уеб услугата методи и поставят резултата в съответните текстови полета.

Ще тестваме приложението, като стартираме две негови инстанции чрез натискане на **[Ctrl+F5]** от VS.NET два пъти последователно. Като натискаме последователно бутона за **Next Session Counter** на двата прозореца, виждаме, че стойностите нарастват последователно и в двата прозореца. Обаче, ако направим същото с бутона **Next Application Counter**, стойностите не нарастват последователно. Всяка стойност върната от **GetApplicationCounter()** е с единица по-голяма от предишната, независимо кой извиква метода. Причината е в това, че този метод ползва **Application** обекта, който е общ за всички инстанции на уеб услугата:



Cookieless сесии към уеб услуга

При уеб услуга със сесии без Cookies ситуацията става по-сложна. Това, за което по принцип се използват Cookies, е в тях да се съхранява уникален идентификатор на сесията. При уеб приложенията, ако сесията не използва Cookies, този идентификатор се пренася чрез URL адреса на приложението. При извикването на уеб услугата от клиентско приложение се изпълнява проста HTTP заявка, в резултат на което уеб услугата съвсем нормално приема, че приложението в същност е Browser и в отговор връща стандартен HTTP отговор "302 Found", а не "200 OK", както очаква

клиентското приложение. В резултат на това се хвърля **WebException** изключение.

Как да решим този проблем? Една от възможностите е да прихващаме изключението и го обработваме по подходящ начин:

```
private Uri mWebServiceUrl;

// Some Code

private void buttonNextSessionCounter_Click(object sender,
   System.EventArgs e)
{
   if(mWebServiceUrl == null)
   {
      mWebServiceUrl = new Uri(mSessionService.Url);
   }
   else
   {
      mSessionService.Url = mWebServiceUrl.AbsoluteUri;
   }
   int nextValue = 0;
   try
   {
      nextValue = mSessionService.GetSessionCounter();
      textBoxNextSessionCounter.Text = nextValue.ToString();
   }
   catch(WebException webException)
   {
      HttpWebResponse httpResponse =
         webException.Response as HttpWebResponse;
      if ( httpResponse != null )
      {
         if(httpResponse.StatusCode == HttpStatusCode.Found)
         {
            mWebServiceUrl = new Uri(mWebServiceUrl,
               httpResponse.Headers["Location"]);
             buttonNextSessionCounter_Click(sender, e);
         }
         else
         {
            throw webException;
         }
      }
      else
      {
         throw webException;
      }
   }
}
```

За да решим проблема използваме една външна променлива **mWebServiceUrl**, в която държим променения адрес на уеб услугата. В началото на метода се проверява дали тази променлива е **null** и ако е, тя се създава с текущия адрес на уеб услугата. Ако не е, то уеб услугата не използва адреса си по подразбиране, а друг, зависещ от идентификатора на сесията на услугата. Тогава, адресът на уеб услугата, се заменя с адреса от променливата **mWebServiceUrl**. Хвърленото изключение има свойство **Response**, което държи HTTP отговора, върнат от уеб сървъра. Ако в действителност се окаже, че отговорът е "302 Found", т. е. неговият **StatusCode** е **HttpStatusCode.Found**, тогава от него се взема новият адрес и се извиква отново функцията.

Разбира се, това решение **не е стандартно и трябва да се избягва!** То е породено от създадения проблем. Принципно рядко се използват сесии в уеб услугите, които да са cookieless.

Ръчна реализация на сесии чрез параметри

Съществува възможност и ръчно да се реализира управлението на сесията. Това може да стане, като се дефинира допълнителен уеб метод **CreateSession()** в уеб услугата, който да връща уникален идентификатор на нова сесия. След това този идентификатор може да се изисква да бъде подаван като параметър при извикването на всички останали методи и той да се използва като ключ в **Application** обекта за съхранение на данните от сесията на различните потребители. Трябва, обаче да се помисли и за сигурността – активните сесии трябва автоматично да се изтриват при продължителна липса на активност (примерно 5 минути), трябва да бъдат достатъчно случайно генерирани, за да бъде трудно откриването им и т.н. Този подход ще демонстрираме в практическия проект в последната тема от книгата (вж. "Практически проект").

Сигурност на уеб услугите

Уеб услугите се използват в много различни ситуации. От съвсем прости двуслойни приложения, състоящи се от едно клиентско приложение, което да осигурява потребителския интерфейс и уеб услуга, която реализира цялата бизнес логика и достъпа до базата, до съвсем сложни архитектури, изградени изцяло на базата на уеб услуги, взаимодействащи една с друга.

Както всяка една услуга, вие можете да предложите вашите уеб услуги на ваши клиенти през Интернет. Достъпът в такъв случай може да бъде позволен само срещу заплащане. В такива ситуации сигурността е изключително важна. Проблемът как да ограничим достъпа до някои уеб методи само за оторизирани потребители има няколко решения. Нека ги разгледаме и обясним техните силни и слаби страни.

Сигурност чрез SSL/HTTPS

Един от простите начини да защитим надеждно цялата комуникация, извършвана между уеб услуга и клиентско приложение, е да използваме

криптиран канал (SSL tunnel) за пренасяните данни. Така отговорността за автентикацията и за криптирането на трафика не е на уеб услугата, а на уеб сървъра, върху който тя е публикувана.

Този подход е лесен за имплементация и осигурява много високо ниво на сигурност. Възможно е да се използва автентикация с цифров сертификат, както от страна на сървъра пред своите клиенти, така и от страна на клиентите пред сървъра. Един от проблемите е, че цифровите сертификати струват скъпо, а ако се използват саморъчно подписани (self-signed) сертификати, сигурността може да бъде компрометирана.

Сигурност чрез предаване на допълнителни параметри

Този подход е свързан с подаване на допълнителни параметри към уеб метода. Това може да са потребителско име и парола, уникален за потребителя идентификатор и др. При извикване на всеки уеб метод, той ще трябва да проверява дали подадените му параметри са коректни. Този подход може да се прилага, когато уеб методите, изискващи оторизиран достъп, не са много на брой, но крие известни рискове. Тъй като съобщенията, които се предават са в XML и няма криптиране, тогава става много лесно някой да прочете тези допълнителни данни, които се подават към уеб метода.

Възможно е посоченият метод да се модифицира, така че паролата да не пътува в чист вид, а автентикацията да се извършва по сигурен начин по схемата "Challenge/Response" (вж. http://en.wikipedia.org/wiki/Challenge-response_authentication).

Възможно е също автентикацията да се извършва еднократно, при което клиентът да получава специален идентификатор (ticket), с който да се автентикира след това пред уеб методите (да го подава като допълнителен параметър на всеки уеб метод). Този идентификатор може да служи едновременно и за поддръжка на клиентска сесия.

Сигурност чрез сесии

Сигурността чрез сесии е един значително по-удобен за прилагане метод, отколкото чрез допълнителни параметри. При него автентикацията на потребителя не става при всяко извикване на всеки уеб метод, а само при извикване на конкретен уеб метод (например **`Login()`**), отговарящ точно за това. Той проверява дали потребителското име и паролата са валидни и ако е така, отбелязва това в ASP.NET сесията (**`HttpSessionState`** обекта) по някакъв начин, например като постави в нея потребителското име под някакъв ключ. Когато се извика някой уеб метод от потребителя, този уеб метод проверява дали в сесията е отбелязана автентикацията на потребителя и ако е така продължава изпълнението си.

За разлика от първия метод, този не изисква всеки път да се подават допълнителни параметри за автентикация, което доста улеснява клиентското приложение.

От гледна точка на сигурността, обаче, не е кой знае колко по-сигурно. Фактът, че не при всяко извикване на метод се подава конфиденциална информация, до някъде затруднява достъпа до нея на хакери, но не решава проблема. При извикване на основния метод за автентикация все пак се предават в чист текст потребителското име и парола.

При автоматично управление на сесията е много по-трудно да се реализира Challenge/Response схемата за автентикация, която защитава паролата на клиента от "подслушване по пътя".

Сигурност чрез сесии – примерна услуга

За да демонстрираме този подход, ще направим съвсем проста уеб услуга, служеща си с ASP.NET сесията за осигуряване на достъп с автентикация до някои от уеб методите:

```
const string USER_NAME_SESSION_KEY = "UserName";

[WebMethod(EnableSession=true, Description =
   "Checks given credentials and establishes a session.")]
public void Login(string aUserName, string aPassword)
{
   if (IsLoginValid(aUserName, aPassword))
   {
      Session[USER_NAME_SESSION_KEY] = aUserName;
   }
   else
   {
      throw new AuthenticationException("Ivalid credentials!");
   }
}

[WebMethod(EnableSession=true, Description =
   "Terminates the active session (if any).")]
public void Logout()
{
   Session.Abandon();
}

[WebMethod(EnableSession=true, Description =
   "Returns some protected data. Requires active session.")]
public string GetProtectedData()
{
   CheckSecurity();
   return "This data is protected!";
}

[WebMethod(Description="Returns some data. Does not require a
session.")]
public string GetNotProtectedData()
{
```

```
   return "This data is not protected.";
}

private void CheckSecurity()
{
   string currentUser = (string) Session[USER_NAME_SESSION_KEY];
   if (currentUser == null)
   {
      throw new AuthenticationException(
        "Access denied! Please login first!");
   }
}

private bool IsLoginValid(string aUserName, string aPassword)
{
   // Just for the demo check if the user and password are equal
   return (aUserName == aPassword);
}

public class AuthenticationException : ApplicationException
{
   public AuthenticationException(string aMessage) :
     base(aMessage) {}
}
```

В началото декларираме една константа от тип **string**, която ще ползваме като ключ, под който ще записваме в сесията името на текущия автентикиран потребител. Основният метод за автентикация е **Login(…)**. Той извиква метода **IsLoginValid(…)** с подадените му потребителско име и парола. В зависимост какво върне **IsLoginValid(…)** или се поставя в сесията потребителското име или се хвърля **AuthenticationException**.

Методът **IsLoginValid(…)** е изключително опростен. Той само проверява дали потребителското име съвпада с паролата. В реална ситуация проверката може да се извърши в база данни или по друг начин.

В горния пример има два метода, които демонстрират достъпа до защитени методи и до незащитени методи. В началото на метода **GetProtectedData()** се проверява дали потребителят има право на достъп. Методът **CheckSecurity()** в действителност проверява дали обекта в сесията с ключ "**UserName**" е **null**. Ако е така, се хвърля **AuthenticationException**. За да се имплементира функционалността на излизане от системата, в примера има метод **Logout()**, който премахва активната ASP.NET сесия.

Сигурност чрез сесии – примерен клиент за услугата

Нека сега да построим едно съвсем елементарно Windows Forms клиентско приложение, демонстриращо работата на представената по-горе уеб услуга. Потребителският му интерфейс ще представлява две текстови полета,

в които ще се въвеждат потребителско име и парола, и четири бутона – по един за всеки уеб метод на услугата. При стартиране на приложението, при зареждане на главната му форма, се инстанцира променлива за уеб услугата и й се присвоява контейнер за cookies:

```
private void MainForm_Load(object sender, System.EventArgs e)
{
   // Instantiate the Web Service proxy class
   mSecuredService = new Services.SecuredService();

   // Add cookies container to the service proxy
   mSecuredService.CookieContainer =
      new System.Net.CookieContainer();
}
```

При натискане на бутона **[Login]** се вземат стойностите на текстовите полета, извиква се уеб методът **Login(…)** и ако изпълнението му мине успешно, се извежда съобщение за успех. В противен случай се извежда съобщение за грешка:

```
private void buttonLogin_Click(
   object sender, System.EventArgs e)
{
   string user = textBoxUserName.Text;
   string pass = textBoxPassword.Text;
   try
   {
      mSecuredService.Login(user, pass);
      MessageBox.Show("Login successfull.", "Info");
   }
   catch (SoapException se)
   {
      MessageBox.Show(se.Message, "Error");
   }
}
```

При натискане на бутона **[Get Protected Data]** преди да се е автентикирал успешно клиентът, излиза съобщение за грешка. Ако клиентът, обаче се е автентикирал преди това, уеб методът се изпълнява успешно и върнатият от него резултат се визуализира:

```
private void buttonGetProtectedData_Click(
   object sender, System.EventArgs e)
{
   try
   {
      string data = mSecuredService.GetProtectedData();
      MessageBox.Show(data, "Info");
   }
```

```
   catch (SoapException se)
   {
      MessageBox.Show(se.Message, "Error");
   }
}
```

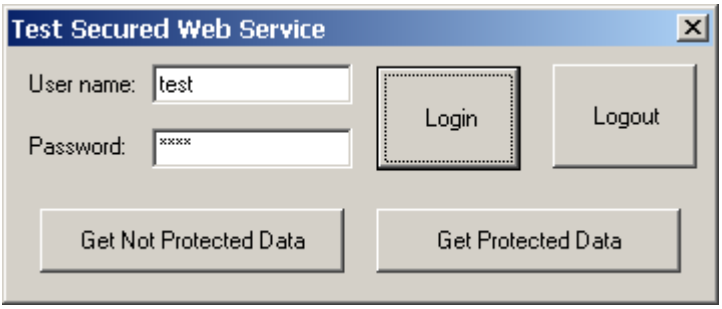
При натискане на бутона **[Get Not Protected Data]** се извиква съответният уеб метод и се извежда съобщение с резултата:

```
private void buttonGetNotProtectedData_Click(
   object sender, System.EventArgs e)
{
   string data = mSecuredService.GetNotProtectedData();
   MessageBox.Show(data, "Info");
}
```

Бутонът **[Logout]** извиква уеб метод, който прекратява активната сесия:

```
private void buttonLogout_Click(
   object sender, System.EventArgs e)
{
   mSecuredService.Logout();
   MessageBox.Show("Logout successfull.", "Info");
}
```

Ето как изглежда клиентското приложение в действие:



Сигурност чрез средствата на Web Service Enhancements (WSE)

Този подход ни предлага най-добро обезпечаване на сигурността при изграждане и използване на уеб услуги. WSE е голяма библиотека от класове, отговорни за приемането и предаването на SOAP пакети между приложенията и уеб услугите. WSE всъщност представлява разширение на .NET Framework и може безплатно да се изтегли от сайта на Майкрософт. След инсталация се интегрира във VS.NET. Очаква се в бъдещи версии да бъде добавен като стандартна част от .NET Framework и VS.NET.

По отношение на сигурността WSE предлага множество от начини за защита на предаваната информация. Чрез WSE много лесно може да се

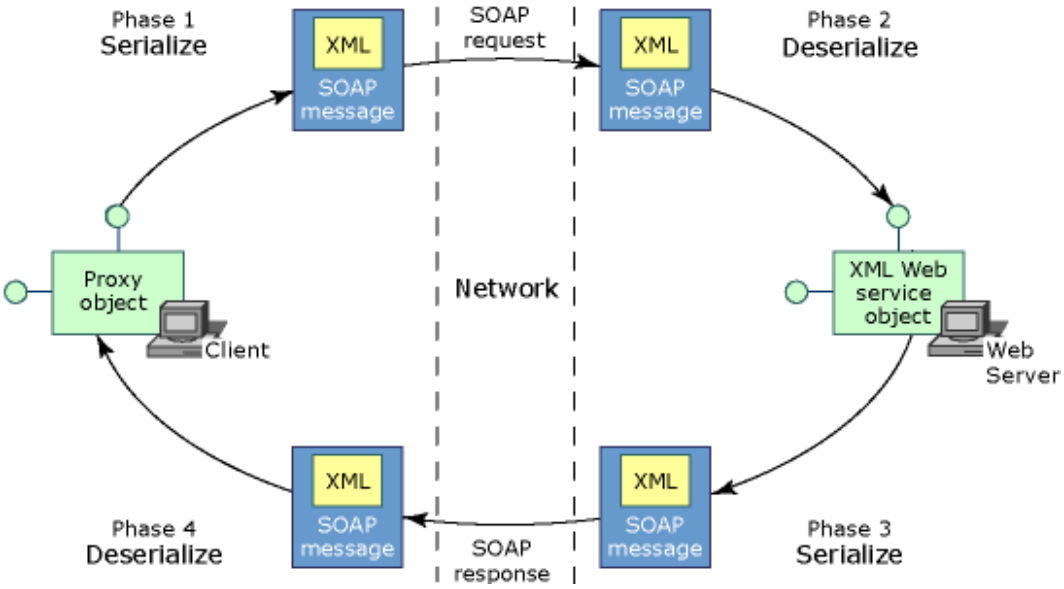
реализира криптиране на трафика, цифрово подписване, автентикация с парола и чрез цифрови сертификати без да се налага да се пише код за това.

Изключенията в уеб услугите

Обработката на изключенията в уеб услугите не е тривиална. Както вече разгледахме, SOAP стандартът дефинира начин за указване на възникналата при изпълнение на услугата грешка, като информацията за нея се поставя в елемента **`fault`** на SOAP тялото.

Жизненият цикъл на едно SOAP съобщение

За да си изясним как точно ASP.NET обработва изключенията, нека да се спрем малко по-подробно на жизнения цикъл на SOAP съобщенията:



На фигурата може да проследим жизнения цикъл на едно SOAP съобщение. Фазите, през които минава то, са следните:

- Фаза 1 – прокси класът, разположен при клиента, сериализира направената заявката във валидно SOAP съобщение и го изпраща към сървъра.
- Фаза 2 – ASP.NET десериализира полученото съобщение и изпълнява съответния метод от уеб услугата.
- Фаза 3 – полученият резултат от изпълнението на услугата се сериализира в SOAP съобщение и се изпраща към клиента.
- Фаза 4 – отново прокси класът получава съобщението, десериализира го и подава получения резултат на извикващия метод от клиентското приложение.

Всички изключения се заместват с SoapException

Когато на сървъра възникне изключение, ASP.NET автоматично го прихваща и го обработва вътрешно. Този процес включва конструиране на валидно SOAP съобщение и сериализиране на информация за възникналата грешка във **`fault`** елемента на SOAP пакета.

Когато съобщението стигне до клиента, прокси класът автоматично парсва неговото тяло и спрямо информацията във **`fault`** елемента конструира **`SoapException`**.

Всичко изглежда идеално на пръв поглед, но както и в реалния живот, всъщност не е така. При конструирането на **`SoapException`** единственото, което се запазва от възникналото изключение на сървъра, е свойството му **`Message`**, което съдържа единствено и само текстово описание на грешката. Всякаква друга информация за изключението е безвъзвратно изгубена – губи се както оригиналният тип на изключението, така и вложените изключения (свойството **`InnerException`** има стойност **`null`**).

Например, ако даден уеб метод хвърли **`DivideByZeroException`**, клиентът вместо да получи изключение от същия тип, получава **`SoapException`**. Това сериозно възпрепятства използването на пълната мощ на изключенията като съвременно средство за обработка на грешки и проблемни ситуации.

Решението на проблема с изключенията

Не си мислете, че ще оставим този проблем отворен. Напротив, ще ви предложим две негови решения.

Първото решава проблема по малко заобиколен начин (workaround), като прави услугата и клиента зависими. За разлика от него, второто предоставя прозрачен начин за запазване на възникналото изключение, но самото е по-сложно и изисква добро познаване на уеб услугите и технологиите, свързани с тях.

Решение 1 – чрез код за грешка

Ще използваме следния подход: Нашата цел е да върнем информация към клиента, ако на сървъра възникне проблем. За целта дефинираме изборен тип (**`enum`**), който съдържа всички възможни грешки, които могат да възникнат при изпълнението на даден уеб метод.

След това в уеб метода, където очакваме да възникне изключението, го прихващаме и връщаме съответния му код на грешка – инстанция на изброения тип, който сме дефинирали.

Когато съобщението пристигне при клиента, той може да провери кода на грешката и да извърши някакво действие спрямо него. Няма да се спираме подробно на това решение, защото неговата реализация може да се разгледа подробно в практическия проект (вж. темата "Практически проект").

Ще отбележим само някои от недостатъците на този подход. Основният от тях е, както вече споменахме, че клиентът и услугата стават тясно зависими, защото клиентът е длъжен да проверява всеки път възникналия код за грешка. Друг недостатък е загубата на всякакви вложени изключения. Много сериозен проблем е липсата на възможност за дефиниране код на грешка за изключения от общ тип, като например: **ArithmeticException**, **ArgumentException**, **IndexOutOfRangeException** и др. Губи се и възможността грешките да се обработват на много нива. На практика този подход ни връща в епохата на процедурното програмиране, при което функциите връщат код на грешка.

Решение 2 – чрез SOAP разширение

Второто решение преодолява проблемите на първото, като се възползва от разширяемата структура на SOAP стандарта и предоставените ни за целта класове от .NET Framework.

Накратко идеята, която ще реализираме е следната: с помощта на разширение тип **SoapExtension** ще прихванем получените в уеб услугата изключения и ще ги сериализираме в изходящото SOAP съобщение.

При клиента ще дефинираме **SoapInputFilter** (входящите и изходящите филтри са една от функционалностите предоставени ни от WSE), който ще десериализира изключението и ще го хвърля локално в клиентското приложение. Нека анализираме проблемите, свързани с реализацията на тази идея, и предложим конкретна имплементация.

Сървърна част

Както вече споменахме, на сървъра ще използваме **SoapExtension**, благодарение на който ще сериализираме изключението. Класът **SoapExtension** ни предлага функционалността да разширим SOAP съобщението, като се намесим в различните стадии от неговата обработка на сървъра.

Етапите, които можем да прихванем, са следните: **BeforeDeserialize**, **AfterDeserialize**, **BeforeSerialize** и **AfterSerialize**, като името на всеки указва кога точно се изпълнява (припомнете си картинката с жизнения цикъл на SOAP съобщението). Ето примерен код, който дефинира нашето разширение **SerializedExceptionExtension** и осигурява сериализиране на възникналото изключение:

```
public class SerializedExceptionExtension : SoapExtension
{
   Stream mOldStream;
   Stream mNewStream;

   public override Stream ChainStream(Stream aStream)
   {
      mOldStream = aStream;
      mNewStream = new MemoryStream();
```

```
    return mNewStream;
  }

  public override void ProcessMessage(SoapMessage aMessage)
  {
    if (aMessage.Stage == SoapMessageStage.AfterSerialize)
    {
      mNewStream.Position = 0;

      if (aMessage.Exception != null)
      {
        if (aMessage.Exception.InnerException != null)
        {
          InsertDetailIntoOldStream(
            aMessage.Exception.InnerException);
        }
      }
      else
      {
        CopyStream(mNewStream, mOldStream);
      }
    }
    else if (aMessage.Stage ==
      SoapMessageStage.BeforeDeserialize)
    {
      CopyStream(mOldStream, mNewStream);
      mNewStream.Position = 0;
    }
  }

  private void InsertDetailIntoOldStream(Exception aException)
  {
    XmlDocument doc = new XmlDocument();
    doc.Load(mNewStream);
    XmlNode detailNode = doc.SelectSingleNode("//detail");

    try
    {
      detailNode.InnerXml =
        GetSerializedExceptionXmlElement(aException);
    }
    catch (Exception exception)
    {
      // Unable to serialize the exception
      detailNode.InnerXml = exception.Message;
    }

    XmlWriter writer =
      new XmlTextWriter(mOldStream, Encoding.UTF8);
    doc.WriteTo(writer);
```

```
      writer.Flush();
   }

   private string GetSerializedExceptionXmlElement(
      Exception aException)
   {
      StringWriter stringWriter = new StringWriter();
      XmlWriter xmlWriter = new XmlTextWriter(stringWriter);

      xmlWriter.WriteStartElement("Serialized");
      xmlWriter.WriteString(SerializeException(aException));
      xmlWriter.WriteEndElement();

      return stringWriter.ToString();
   }

   private string SerializeException(Exception aException)
   {
      MemoryStream stream = new MemoryStream();
      IFormatter formatter = new SoapFormatter();
      formatter.Serialize(stream, aException);
      stream.Position = 0;
      return Encoding.UTF8.GetString(stream.GetBuffer());
   }

   private void CopyStream(Stream aFrom, Stream aTo)
   {
      TextReader reader = new StreamReader(aFrom);
      TextWriter writer = new StreamWriter(aTo);
      writer.WriteLine(reader.ReadToEnd());
      writer.Flush();
   }

   // SoapExtension methods implementation
   public override object GetInitializer(LogicalMethodInfo
      aMethodInfo, SoapExtensionAttribute aAttribute)
   {
      return null;
   }

   public override object GetInitializer(Type aServiceType)
   {
      return null;
   }

   public override void Initialize(object aInitializer)
   {
   }
}
```

За да дефинираме клас, който може да се използва като разширение (extension) на SOAP, трябва да наследим абстрактния клас **SoapExtension**, който се намира в пространството от имена (namespace) **System.Web.Services.Protocols**. Наследявайки **SoapExtension** трябва да припокрием (**override**) абстрактните му методи: **Initialize(…)**, **GetInitializer(…)** и **ProcessMessage(…)**. Първият от тях (той има две декларации с различни параметри) се използва за инициализиране на данни, които ще се използват вътрешно в SOAP разширението. В нашия случай няма да го използваме. Вторият метод **ProcessMessage(…)** е най-съществения. В него се извършва обработката на съобщението.

Преди да преминем към по-детайлното му разглеждане, нека да обърнем внимание на метода **ChainStream(…)**. Той е деклариран като виртуален в **SoapExtension** и е единственият начин, чрез който можем да получим поток към текущото SOAP съобщение. Тъй като искаме да модифицираме този поток, трябва да запазим референция към него (**mOldStream**) и да направим нов поток (**mNewStream**), в който ще запишем нашето модифицирано съобщение.

Да преминем към ключовия метод **ProcessMessage(…)**. В него трябва да направим две неща: първо, ако етапът от обработката, в който се намира съобщението, е **BeforeDeserialize** (току що получено и още не десериализирано), копираме входящия поток (**mOldStream**) в конструирания нов поток (**mNewStream**) чрез помощния метод **CopyStream(…)**.

Ако етапът от обработката е **AfterSerialize** (изходящото съобщение е сериализирано и е готово за изпращане към клиента), проверяваме за възникнало изключение. Ако няма такова, просто копираме новия поток обратно в стария. Ако пък е възникнало изключение, намираме **detail** елемента в SOAP грешката (**fault**), която се е генерирала, и в него сериализираме изключението (припомнете си, че **detail** елемента може да съдържа XML). Отново, вече промененото съобщение записваме в стария поток.

Обърнете внимание, че използваме бинарна сериализация и класа **SoapFormatter**, а не **XmlSerializer**. Бинарната сериализация позволява да се сериализират не само публичните полета на изключението, а цялото му състояние (включително и частните вътрешни полета). SOAP форматерът позволява резултатът от сериализацията да е във вид на XML.

Класът **System.Runtime.Serialization.Formatters.Soap.SoapFormatter** е дефиниран в асемблито **System.Runtime.Serialization.Formatters.Soap.dll** и преди да се използва трябва да добавим ръчно към проекта референция към това асембли.

Имайте предвид, че при сериализацията на изключението, всички негови свойства се предават в текстов вид. Ако тази информация е конфиденциална, тялото на SOAP съ-

	общението може да се криптира.

След като нашето SOAP разширение е готово, нека да разгледаме по какъв начин може да го приложим в уеб услугата. Вариантите са два: чрез атрибути да укажем към кои уеб методи да се прилага или чрез конфигурационна настройка да го приложим върху всички уеб методи. Ще разгледаме и двата варианта.

Настройка в Web.config файла

За да добавим SOAP разширението към всички уеб методи, трябва да добавим следните редове в конфигурационния файл на ASP.NET уеб услугата (**`Web.config`**), в секцията **`system.web`**:

```
<webServices>
  <soapExtensionTypes>
    <add type="SerializedExceptionExtension, MyWebService"
      priority="1" group="0" />
  </soapExtensionTypes>
</webServices>
```

Атрибутът **`type`** указва, кой е класът, който ще се използва, и в кое асемблы се намира той. Другите два атрибута **`priority`** и **`group`** указват приоритета и последователността на изпълнение на SOAP разширенията (ако има повече от едно).

Настройка чрез атрибут

Нека да разгледаме и втория начин за прилагане на SOAP разширението, което дефинирахме. Този вариант е по-гъвкав, защото ни дава възможността да приложим разширението само върху конкретни уеб методи, а не върху всички. Дефинираме клас (потребителски атрибут), наследник на **`SoapExtensionAttribute`**, който да припокрие неговите абстрактни свойства **`Property`** и **`ExtensionType`**. Ето неговият сорс код:

```
[AttributeUsage(AttributeTargets.Method)]
public class SerializedExceptionExtensionAttribute :
  SoapExtensionAttribute
{
  public override Type ExtensionType
  {
    get
    {
      return typeof(SerializedExceptionExtension);
    }
  }

  public override int Priority
  {
```

```
      get { return 0; }
      set {    }
   }
}
```

На по-късен етап, ако приложим към даден уеб метод този атрибут, той ще включи за него SOAP разширението.

Ето накрая и кода на уеб метода върху, към който сме приложили атрибута **[SerializedExceptionExtension]**. В този метод съвсем умишлено предизвикваме изключение **DivideByZeroException**, за да илюстрираме неговата сериализация и предаване към клиента на уеб услугата:

```
[WebMethod]
[SerializedExceptionExtension]
public int ThrowException()
{
   int zero = 0;
   return 100 / zero;
}
```

Тестване на SOAP разширението

Ако сега сложим точка на прекъсване (breakpoint) в нашето SOAP разширение и извикаме уеб метода на услугата през тестовата страница на услугата от Internet Explorer, SOAP Extension класът няма да се изпълни. Причината за това е в начина, по който се извиква услугата.

При извикване на уеб услуга през нейната тестващата уеб страница (която се показва при стартиране услугата от VS.NET) уеб методите се изпълняват с директна HTTP GET заявка, а не чрез SOAP заявка. Това е причината SOAP разширенията да не се изпълняват при извикването на уеб методи по този начин.

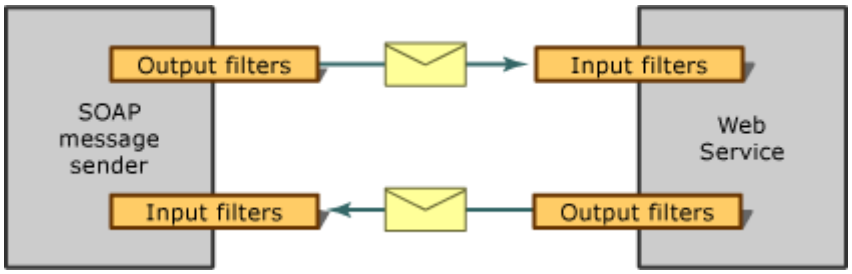
Ако искаме да тестване уеб услугата заедно с прикаченото към нея SOAP разширение, трябва да напишем клиентско приложение, което я консумира.

Клиентска част

След като вече сме сериализирали възникналото изключение във валидно SOAP съобщение, нека да разгледаме какво ни трябва при клиента, за да направим целия процес напълно прозрачен. Нашата цел е да прихванем съобщението, преди то да е стигнало при клиента, да проверим за възникнала грешка и ако има такава, да я десериализираме и да я хвърлим като локално изключение.

Както вече споменахме, ще използваме една от функционалностите на WSE – входящите филтри. (Повече подробности относно WSE може да

намерите на адрес: http://msdn.microsoft.com/webservices/, където може да се свали и актуалната им версия). На следващата картинка можем да видим опростената архитектура, на която се базират WSE, а именно прилагане на дадени филтри върху изходящите съобщения и обратното им налагане върху входящите.



Ако вече имаме инсталирана версия на WSE, можем да преминем към създаването на нашия филтър. Но преди да преминем към кода, нека първо да разрешим използването на WSE в нашия проект, който в случая е просто конзолно приложение. За целта с десния бутон на мишката щракаме върху проекта и от появилото се контекстно меню избираме **WSE Settings X ...**, където **X** е текущо инсталираната версия на WSE. От появилия се диалогов прозорец маркираме **Enable this project for Web Services Enchantments** и натискаме `[ок]`:

WSE автоматично добавят конфигурационен файл за приложението (ако няма такъв), разполагат своите настройки в него и добавят референция към тяхното асембли в проекта.

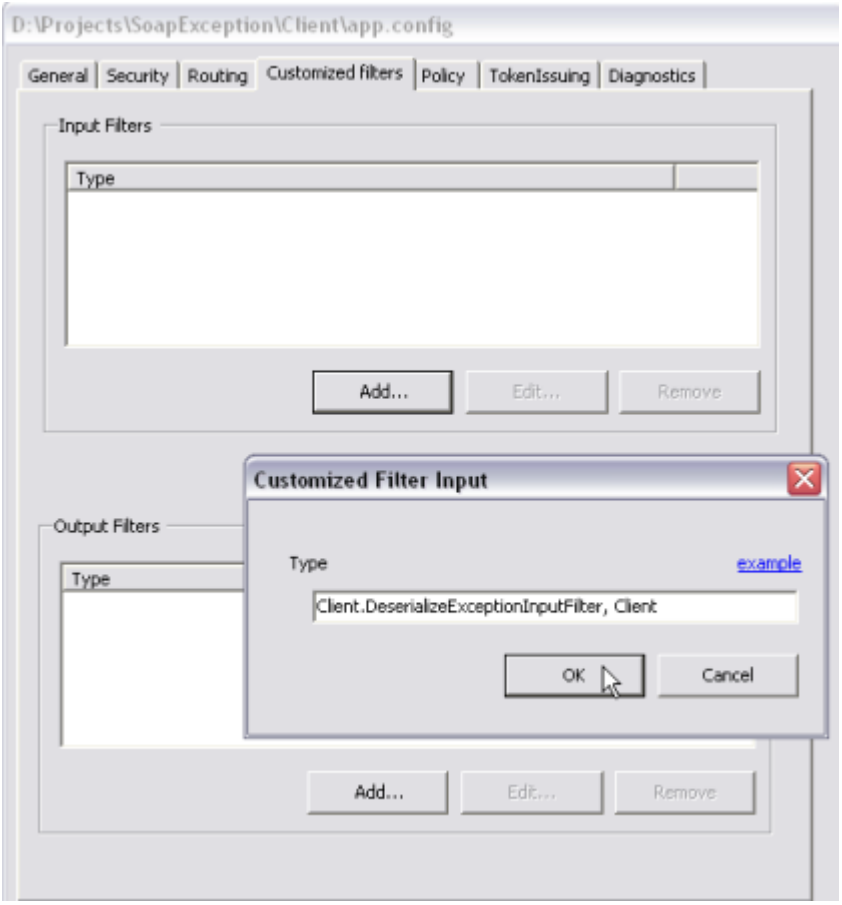
Да преминем към кода на входящия филтър, който ще десериализира възникналото изключение, ако има такова:

```
public class DeserializeExceptionInputFilter : SoapInputFilter
{
   public override void ProcessMessage(SoapEnvelope aEnvelope)
   {
      if (aEnvelope.Fault != null)
      {
         XmlDocument doc = new XmlDocument();
         doc.LoadXml(aEnvelope.InnerXml);

         XmlNode detailNode = doc.SelectSingleNode("//detail");
         if (detailNode != null)
         {
            string serialized =
               GetNodeText(detailNode, "Serialized");
            if (serialized != string.Empty)
            {
               Exception exception =
                  DeserializeException(serialized);
               if (exception != null)
               {
                  throw exception;
               }
            }
         }
      }
   }

   private string GetNodeText(XmlNode aParent, string aNodeName)
   {
      XmlNode node = aParent.SelectSingleNode(aNodeName);
      if (node != null)
      {
         return node.InnerText;
      }
      return string.Empty;
   }

   private Exception DeserializeException(
      string aSerializedException)
   {
      byte[] buffer =
         Encoding.UTF8.GetBytes(aSerializedException);
      MemoryStream stream = new MemoryStream(buffer);

      IFormatter formatter = new SoapFormatter();
      return formatter.Deserialize(stream) as Exception;
   }
}
```

За да може даден клас да се използва като входящ филтър, той трябва да наследява абстрактния клас **`SoapInputFilter`**. Отново ключовият метод е

`ProcessMessage(…)`, като в него проверяваме дали има SOAP грешка. Ако има такава, намираме "**`detail`**" елемента, взимаме съдържанието на неговия поделемент "**`Serialized`**" (споменете си, че като коренов елемент при сериализацията на сървъра, създадохме елемент именно с това име). Съдържанието на елемента всъщност е сериализираното изключение. Десериализираме го и го хвърляме локално в клиентското приложение.

За да добавим така създадения филтър към филтрите на нашия проект отново от менюто на WSE избираме етикета **Customized filters** и добавяме нашия филтър към входящите такива:



Това може да се направи и от конфигурационния файл на приложението (това е файлът **`App.config`** за VS.NET проекти). Нека да разгледаме какво са добавили WSE в него:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <section name="microsoft.web.services2" type=
      "Microsoft.Web.Services2.Configuration.
```

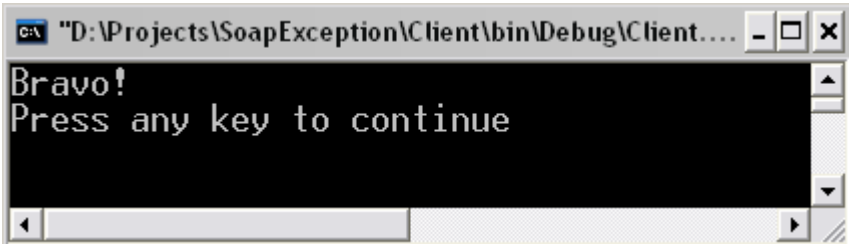
```
      WebServicesConfiguration, Microsoft.Web.Services2,
      Version=2.0.0.0, Culture=neutral,
      PublicKeyToken=31bf3856ad364e35" />
  </configSections>
  <microsoft.web.services2>
    <filters>
      <input>
        <add type="Client.DeserializeExceptionInputFilter,
          Client" />
      </input>
    </filters>
    <diagnostics />
  </microsoft.web.services2>
</configuration>
```

Въпреки, че в горния пример името на класа, който имплементира филтъра и името на асемблито, в което този клас е дефиниран, са на различни редове, това е само защото не се събират на един ред. В реална ситуация стойността на атрибута **type** не трябва да се пренася между редовете.

Остана да разгледаме последната част при клиента – добавяне на референция към уеб услугата. Добавянето не се различава по нищо от обикновеното, единственото по-различно е, че щом WSE са разрешени за проекта, те автоматично генерират нови прокси класове **XxxWse**, където **Xxx** е името на прокси класа създаден от нас. Тези класове са нужни за да могат чрез тях да се наложат дефинираните филтри. Ето и кода на клиента, който използва прокси класа създаден от WSE:

```
class Client
{
   static void Main()
   {
      ExceptionServiceWse service = new ExceptionServiceWse();
      try
      {
         service.ThrowException();
      }
      catch (DivideByZeroException)
      {
         Console.WriteLine( "Bravo!" );
      }
   }
}
```

Ако сме направили всичко успешно трябва да видим дългоочаквания резултат:



Той показва, че клиентът е прихванал **`DivideByZeroException`**, който е подаден от сървъра.

Упражнения

1. Какви модели за разпределени приложения познавате? Каква е разликата между тях?
2. Какво представляват уеб услугите? Какъв проблем решава тази технология?
3. Какво представлява инфраструктурата на уеб услугите? От какво се състои?
4. Какво представляват UDDI директориите и за какво служат?
5. Какво е DISCO и за какво служи?
6. Какво е WSDL и за какво се използва?
7. Какво е SOAP? От какво се състои? За какво се използва?
8. Опишете типични сценарии за използване на уеб услуги при .NET базирани приложения.
9. Без да използвате VS.NET създайте проста уеб услуга (**`.asmx`** файл), която по зададена дата връща деня от седмицата (на български език). Инсталирайте уеб услугата на IIS. Тествайте с Internet Explorer. Разгледайте WSDL описанието.
10. Без да използвате VS.NET създайте клиент (конзолно приложение) за уеб услугата от предходната задача.
11. С помощта на VS.NET създайте уеб услуга, която приема 2 символни низа и връща колко пъти първият се среща във втория. Дефинирайте уникален **`namespace`** за уеб услугата. Задайте подходящо описание на уеб метода й.
12. С помощта на VS.NET създайте клиент за уеб услугата от предходната задача.
13. В едно училище учат ученици, разпределени в различни класове. Всички ученици от даден клас изучават някаква съвкупност от учебни предмети и имат по няколко оценки от изпитванията по всеки предмет. Проектирайте релационна схема, която да съхранява информация за учениците, класовете, учебните предмети и оценките. Реализирайте

уеб услуга, която изпълнява следните операции (чрез SQL команди към БД):

- добавяне/изтриване на клас
- добавяне/изтриване/промяна на ученик, извличане на учениците (от даден клас)
- добавяне/изтриване на учебен предмет, извличане на учебните предмети (за даден клас)
- добавяне/изтриване/извличане на оценки (на даден ученик по даден предмет)

Използвайте свързания модел от ADO.NET.

14. Създайте Windows Forms клиент за уеб услугата от предходната задача. Приложението трябва да визуализира класовете и да позволява навигация сред тях. При избор на даден клас трябва да се показват учениците, които го съставят и учебните предмети, които тези ученици изучават. Трябва да се позволява редактиране на учениците и учебните предмети за текущия избран клас. При избор на ученик трябва да се позволява редактиране на оценките му по всеки от учебните предмети. При всяка редакция трябва да се извиква уеб метод от услугата чрез който измененията да се нанасят в базата данни. При промяна на текущия избран клас, трябва да се извличат наново учениците и предметите. При промяна на избрания ученик трябва оценките му да се зареждат наново от уеб услугата.

15. Създайте уеб услуга, която по зададено цяло число p (p е **`UInt32`**) намира и връща броя прости числа в интервала [1…p]. Услугата би трябвало да работи бавно при големи стойности на p. Създайте Windows Forms приложение, което съдържа текстово поле, бутон и списък. При въвеждане на число в текстовото поле и натискане на бутона трябва да се извиква асинхронно уеб услугата за пресмятане на простите числа между 1 и p. При завършване на асинхронно извикване резултатът трябва да се добавя в списъка във формат "**`Primes in range [1…p] are XXX`**". Не забравяйте, че асинхронните извиквания използват нишки от вградения Thread Pool на .NET Framework, които не трябва да достъпват директно потребителския интерфейс.

16. Проектирайте релационна схема от таблици в MS SQL сървър, която описва потребители и правата им за достъп в дадена система. Всеки потребител се характеризира с име, login и парола и може да има достъп до подмножество от функциите на системата. Всяка функция в системата си има име и може да е достъпна от подмножество на потребителите. Създайте уеб услуга, която чрез използване на несвързания модел на достъп до данни в ADO.NET реализира уеб методи за извличане на данните (във вид на **`DataSet`**) и за обновяване на променени данни (съдържащи се в **`DataSet`**). Реализирайте Windows Forms приложение, което позволява редактиране на потребителите и техните права използвайки уеб услугата.

17. Реализирайте системата за управление на потребители и техните права от предходната задача като добавите автентикация в уеб услугата и защитите методите за достъп до данните чрез ASP.NET сесията. Уеб услугата трябва да позволява достъп до защитените методи само на потребителя с име "admin". Първоначално създайте този потребител директно в базата данни на ръка.

Използвана литература

1. Светлин Наков, Уеб услуги с ASP.NET – http://www.nakov.com/dotnet/lectures/Lecture-20-Web-Services-v1.0.ppt
2. Стоян Йорданов, Уеб услуги – http://www.nakov.com/dotnet/2003/lectures/Web-Services.doc
3. MSDN Training, Developing XML Web Services Using Microsoft® ASP.NET (MOC 2524B)
4. Keith Ballinger, .NET Web Services: Architecture and Implementation, Addison Wesley, 2003, ISBN 0321113594
5. Scott Short, Building XML Web Services for the Microsoft .NET Platform, Microsoft Press, 2002, ISBN 0735614067
6. Damien Foggon, Daniel Maharry, Chris Ullman and Karli Watson , Programming Microsoft .NET XML Web Services, Microsoft Press, 2004, ISBN 0735619123
7. Building the Next Generation of Service-based Software Systems, MSDN Library – http://msdn.microsoft.com
8. The ASP Column - Using SOAP Extensions in ASP.NET, MSDN Library – http://msdn.microsoft.com
9. Consuming a DataSet from an XML Web Service, MSDN Library – http://msdn.microsoft.com
10. MSDN Library – http://msdn.microsoft.com

Глава 22. Отдалечени извиквания с .NET Remoting

Автор

Виктор Живков

Необходими знания

- Базови познания за .NET Framework
- Базови познания за езика C#
- Базови познания по компютърни мрежи и Интернет технологии
- Базови познания по разпределени архитектури и системи
- Познания за сериализацията в .NET Framework

Съдържание

- Какво е .NET Remoting?
- Кога се използва Remoting?
- Remoting инфраструктурата
- Remoting канали и форматери. Регистрация на канал
- Активация на обекти. Активация от сървъра. Активация от клиента
- Маршализация (Marshaling). Marshal-by-Value обекти. Marshal-by-Reference обекти
- Живот на обектите (Lifetime). **`ILease`**
- Remoting конфигурационни файлове
- Практика: Създаване на Remoting сървър и клиент
- Проблемът с общите типове

В тази тема ...

В настоящата тема ще разгледаме инфраструктурата за отдалечени извиквания, която .NET Framework предоставя на разработчиците. Ще обясним основите на Remoting технологията и всеки един от нейните компоненти. Ще започнем с канали и форматери и ще продължим с отдалечените обекти и тяхната активация. Ще се спрем на разликата между различните типове отдалечени обекти, жизнения им цикъл и видовете маршализация. Стъпка по стъпка ще достигнем до създаването на примерен Remoting сървър и клиент. Ще завършим с обяснение на един гъвкав и практичен начин за конфигуриране на цялата Remoting инфраструктура.

Разпределени приложения

Поради сложния характер на днешните приложения по-голямата част от тях са разпределени. Състоят се от няколко отделни компонента, които често пъти са отдалечени един от друг, но за целта на приложението трябва да взаимодействат помежду си. Съществуват няколко утвърдени от практиката модели за разпределени приложения:

- Клиент/сървър – двете страни комуникират помежду си чрез заявки. Клиентът изпраща заявка към сървъра, сървъра я обработва и връща резултата обратно.
- Разпределени обекти (distributed objects) – примери за такива инфраструктури за:
  - DCOM – използва се в Microsoft Windows
  - CORBA – отворен стандарт, който за съжаление е много сложен
  - Java RMI – стандарт базиран на Java технологията
  - .NET Remoting – стандарт предлаган от .NET Framework
- Уеб услуги (Web services) – стандартизирана технология, при която отделни обекти на различни платформи и системи комуникират посредством стандартни SOAP съобщения.

По-нататък в настоящата тема ще разгледаме подробно само .NET Remoting технологията. Няма да правим сравнение между различните модели, такива тъй като тази тема не е обект на настоящата книга. Уеб услугите в .NET Framework са разгледани подробно в главата за уеб услуги.

Какво е .NET Remoting?

Remoting технологията в .NET Framework предлага на разработчика прозрачен достъп до отдалечени обекти, без излишни трудности и загуба на гъвкавост. Подходяща е за случаи, когато се налага да се достъпват обекти, които:

- се намират в друг домейн на приложението (application domain)
- принадлежат на друг процес
- се намират на отдалечена машина

Независимо от местоположението на интересуващите ни обекти Remoting инфраструктурата осигурява лесен начин за използването им. Комуникацията между обектите се извършва чрез стандартизиран механизъм и специалната инфраструктура, осигурени от .NET Framework.

Кога се използва Remoting?

Тъй като Remoting технологията ни позволява да осъществим достъп до и реализация на отдалечени обекти, тази технология може да бъде решение за проблеми, свързани с мрежова комуникация. Самата инфраструктура е гъвкава и разширяема. Тя дава решения на голям набор от проблеми без

особено усилия от страна на програмиста. .NET Framework ни предлага и алтернативи в лицето на уеб услугите и мрежовата комуникация на по-ниско ниво (виж **`System.Net`**). Повече за това, кога е удачно да използваме Remoting, можете да намерите в частта "Remoting сценарии".

Microsoft Indigo (WCF)

Друга зараждаща се технология с голям потенциал е Microsoft Indigo (Windows Communication Framework, WCF). Тя включва в себе си възможностите на ASMX (или т. нар. уеб услуги), .NET Remoting, Enterprise Services, WSE (Web Services Enchancements) и MSMQ (Microsoft Message Queue). В Indigo е доразвита концепцията за комуникация между отдалечени обекти, като се набляга на съвместимостта на различните среди и платформи. Архитектурата на Indigo е пример за имплементация на набиращата популярност Service Oriented Architecture (SOA).

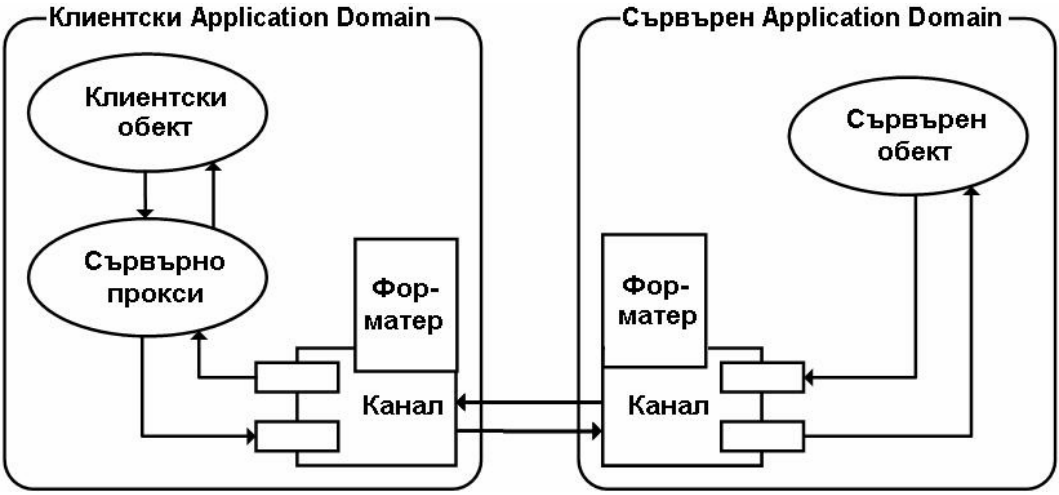
Заради развитието на тази технология в .NET Framework 2.0 се очакват промени, свързани със самия Remoting. Обратната съвместимост е запазена, но Microsoft препоръчват преминаване към WCF (Indigo), след като излезе финалната му версия.

Remoting инфраструктурата

Remoting инфраструктурата се състои от:

- **канали** – канали, по които се пренасят данните и съобщенията от и към отдалечените обекти
- **форматери** – отговарят за форматирането, кодирането и декодирането на съобщенията, които се пренасят чрез каналите в някакъв формат
- **прокси обекти** – предават извикванията на методи към и от отдалечените обекти
- механизми за **активация** – служат за създаване и получаване на инстанция на отдалечения обект
- **маршализация** – осигурява пренос на обекти, техните свойства, полета и т.н.

Като нагледен пример за това каква е взаимовръзката между тези компоненти и как работят те, нека разгледаме следната диаграма:



Как работи Remoting инфраструктурата?

При създаването на инстанция на отдалечен обект Remoting инфраструктурата автоматично създава при клиента прокси клас, който се грижи за обмяната на съобщенията между клиентския и сървърния обект. За клиента съществуването му остава незабележимо, тъй като всяко извикване се обработва автоматично от средата. След като проксито получи контрола на съобщението, го предава на следващия възел в инфраструктурата – канала. Каналът работи тясно свързано с форматер, който форматира данните, обменяни между клиента и сървъра. След това съобщението се изпраща по TCP сокет до получателя, десериализира се, сървърният обект извършва извикването и връща резултата по обратния път към клиента. Всяка от описаните стъпки, с изключение на създаването на сървърно прокси, може да бъде настроена допълнително, така че да отговаря на специфични нужди.

Remoting канали

Каналите се използват за пренос на съобщения от и към отдалечени обекти. Когато клиентът извика метод на отдалечен обект, параметрите, както и всички детайли, свързани с даденото извикване, се транспортират през канал до обекта. Резултатът от извикването се връща обратно при клиента по същия начин. Всеки клиент може да избере към точно кой канал от регистрираните на сървъра да се прикачи, за да комуникира с отдалечения обект. Това дава изключителна гъвкавост на разработчика в избора на канал, така че той да отговаря на специфичните нужди.

За канали могат да се използват стандартните вградени канали, техни модификации и дори напълно нови, създадени от разработчика канали, които да комуникират по избран протокол. В .NET Framework са реализирани три типа канали:

- **TCP канал** – използва се чист TCP сокет. Данните се сериализират стандартно посредством бинарен форматер и се транспортират

посредством TCP протокол. При нужда форматерът може да бъде преконфигуриран като SOAP.

- **HTTP канал** – използва се SOAP протокол. Съобщенията се сериализират чрез SOAP форматер в XML с включени SOAP хедъри. Може да се конфигурира да използва бинарен форматер вместо стандартния SOAP. Комуникацията между обектите се извършва посредством HTTP протокол и се използва модела на заявка/отговор (request / response) подобно на този в уеб приложенията.
- Други канали – дефинирани от разработчика. Те трябва да имплементират интерфейсите: **`IChannel`**, **`IChannelReceiver`** и/или **`IChannelSender`**. Както се вижда от имената на последните два интерфейса, първият служи за канал, който може само да приема съобщения, а вторият – за такъв, който може само да изпраща. В повечето случаи каналите имплементират и двата интерфейса, за да могат да комуникират в двете посоки.

Каналите се използват както от сървърните приложения, така и от клиентските. Каналът трябва да бъде регистриран и от двете страни преди началото на комуникация между тях. За преноса на потока от съобщения се използват TCP портове, през които се подават форматираните данни на извикването.

Регистрация на канал

Регистрацията на канал е задължителна стъпка в подготвянето на комуникацията с отдалечени обекти. Този процес има следните изисквания и ограничения:

- Най-малко един канал трябва да бъде регистриран преди да бъде направено обръщение към отдалечен обект. Каналите задължително се регистрират преди регистрацията на самите обекти.
- Всеки канал се регистрира за даден application domain. В един процес е възможно да има няколко application domains. В момента, в който дадения процес приключи работата си, всички канали, които са регистрирани в него, се унищожават автоматично.
- В рамките на даден application domain всеки канал трябва да има уникално име. Разгледайте примера за да видите начина, по който можете да зададете уникално име на канал:

```
IDictionary properties = new Hashtable();
properties["name"] = "http1";
properties["port"] = "9001";

ChannelServices.RegisterChannel(
  new HttpChannel(properties, null, null));
```

- Не е възможно два или повече канала да бъдат регистрирани върху един и същ порт, по едно и също време, на една и съща машина. От друга страна е възможно един канал да бъде регистриран на повече от един порт.

- В случай, че не сте сигурни кой порт да използвате за канала, можете да накарате Remoting инфраструктурата да ви предостави автоматично свободен порт като регистрирате канал за порт 0.
- Клиентът може да комуникира с отдалечения обект, използвайки който и да е регистриран канал. Remoting инфраструктурата автоматично осигурява свързването на отдалечения обект с правилния канал. Клиентът е задължен да регистрира канал преди да се опита да комуникира с отдалечения обект.

Пример за регистрация на канал

Регистрацията на канал може да се извърши по два начина:

- Чрез класа **`System.Runtime.Remoting.Channels.ChannelServices`**:

```
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

...

TcpChannel channel = new TcpChannel(1234);
ChannelServices.RegisterChannel(channel);
```

- Чрез класа **`System.Runtime.Remoting.RemotingConfiguration`** и конфигурационен файл:

```
RemotingConfiguration.Configure("MyClient.exe.config");
```

Конфигурационните файлове, като средство за настройка на Remoting инфраструктурата, ще разгледаме малко по-късно в настоящата тема.

Форматери (formatters)

Ролята на форматерите в Remoting инфраструктурата е да сериализират съобщенията между двете страни в определен формат. На разположение на разработчиците са два вградени форматера плюс възможността да създадат свой собствен форматер, който да отговаря на специфичните им нужди.

- **SOAP форматер** – сериализира поток в SOAP съвместим XML формат. Използването SOAP протокола позволява съвместимост с множество платформи и клиенти. Поради употребата на XML, обемът на предаваните данни е голям и това води до намаляване на производителността.
- **Бинарен форматер** – сериализира поток в двоичен формат, който е с много по-малък обем от SOAP варианта. Поради вида на сериализирания поток не е толкова съвместим с различни системи, колкото е SOAP потока.

- Друг форматер – форматер, реализиран от разработчика за неговите специфични цели и нужди.

Форматери по подразбиране

Всеки от предоставените от .NET Framework канали има форматер по подразбиране. На разработчика се предоставя готово решение със системата от класове за осъществяване на отдалечените извиквания и той може да се съсредоточи върху реализацията на програмната логика. Двата канала, които са вградени в Remoting инфраструктурата, имат форматери по подразбиране:

- TCP каналът има бинарен форматер.
- HTTP каналът има SOAP форматер.

Тези форматери могат да се преконфигурират. Ето един пример как можете да реализирате в своето приложение TCP канал със SOAP форматер:

```
SoapServerFormatterSinkProvider provider =
  new SoapServerFormatterSinkProvider();
IDictionary properties = new Hashtable();
properties["port"] = 12345;

TcpChannel channel = new TcpChannel(properties, null, provider);
```

Друг начин за промяна на стандартния форматер на вградените канали е чрез редактиране на конфигурационния файл **machine.config**. Този начин е разгледан подробно в частта описваща конфигурирането на Remoting инфраструктурата в края на темата.

Активация на обекти

Процесът на създаване на инстанция на отдалечен обект се нарича **активация**. Remoting инфраструктурата предоставя два начина за извършването й:

- **Сървърна активация** (server-side activation) – използва се в случаи, когато няма необходимост отдалечените обекти да поддържат своето състояние между две извиквания (**SingleCall**) или когато множество клиенти извикват методи на една и съща инстанция на обекта и трябва да се поддържа състоянието на обекта между извикванията (**Singleton**).
- **Клиентска активация** (client-side activation) – инстанцията на обекта е създадена само за клиента, който я е извикал. Само той я управлява и определя времето й за живот. Жизненият цикъл на този вид отдалечени обекти се управлява чрез **lease-based** система на инфраструктурата.

Преди да е възможна активацията на обекти, е задължително да бъдат регистриран всеки от типовете на отдалечените обекти, които смятаме да използваме.

SingleCall режим на активация

SingleCall режимът е приложим при обекти със сървърна активация. Характерно за този режим е, че при всяко извикване на метод от страна на клиента се създава нова инстанция на обекта на сървъра, която обслужва извикването и веднага след това завършва своя жизнен цикъл.

Такива обекти са удобни в случай, че имаме метод, който трябва да свърши определена работа, за извършването, на която нямаме нужда от предишното състояние (state) на обекта. Обекти с такъв режим на работа улесняват балансирането на натоварването (load-balancing) на сървъра, тъй като не се налага да се поддържа състоянието им между отделните извиквания.

Singleton режим на активация

Singleton режимът е приложим при обекти със сървърна активация. Характерно при него е, че съществува единствена инстанция на отдалечения обект и всички клиенти работят с нея. Нейните данни, методи и свойства са споделени между всички клиенти. Самият обект поддържа своето състояние (state).

Този режим на активация е подходящ, когато данните на обекта, трябва да бъдат споделени или когато инстанцирането и поддържането на състоянието на обекта създава допълнително натоварване. При използването на Singleton обекти е възможно няколко клиента едновременно да работят с тях, поради което трябва да се осигури синхронизация при достъпа до общи данни (вж. главата за нишки и синхронизация).

Клиентска активация

При активация от страна на клиента, след заявката на сървъра се създават инстанции на отдалечения обект. Създаденият обект обслужва единствено своя клиент. Обектът може да поддържа своето състояние между извикванията. Жизненият цикъл на всяка инстанция се контролира чрез **lease-based** системата на Remoting инфраструктурата (ще я разгледаме подробно в частта "Живот на обектите".

При всяка заявка за създаване на нов обект се връща нова инстанция. С такъв тип обекти трябва да се работи внимателно, тъй като е възможно за един клиент на сървъра да има повече от една инстанция. При проектиране на такива класове трябва да се отдели време за оптимизиране на методите на класа и да се направи преглед на това колко и какви данни съдържа той, за да се облекчи натоварването на сървъра.

Регистрация на отдалечен обект

Регистрацията на отдалечените обекти следва веднага след регистрацията на Remoting канали. За да извършим това ни е необходимо да предоставим на Remoting инфраструктурата следните данни:

- името на асемблито, в което се намира класа, който искаме да регистрираме
- типа на отдалечения обект
- URI (Unique Resource Identifier) на обекта, чрез който клиентът да може да се обръща към него
- в случай, че обектът е със сървърна активация – неговия режим на активация (**SingleCall** или **Singleton)**

Отново имаме избор между два начина за регистриране – чрез код или чрез конфигурационен файл.

Чрез регистрацията предоставяме данни, които описват уникално на Remoting инфраструктурата всеки тип, който искаме да използваме като отдалечен. Този процес на регистрация е задължителен както за сървъра така и за клиента. Важен момент при регистрирането на типовете е регистрациите при сървъра и клиента да си съответстват. Друго изискване е, типовете, които се използват за отдалечени обекти на сървъра и тези на клиента да имат една и съща версия или поне техните интерфейси да съвпадат. Този конкретен проблем е разгледан в частта "Проблемът с общите типове".

Регистрация на сървъра (Server-Side Registration)

Чрез програмен код имаме три възможности за регистриране на типа на обект, който искаме да е достъпен отдалечено. Използват се методите на класа **`RemotingConfiguration`**:

- **`RegisterWellKnownServiceType(…)`** с параметър, указващ че обектът трябва да се използва в **`SingleCall`** режим – регистрира тип на отдалечен обект в режим **`SingleCall`**. Пример:

```
RemotingConfiguration.RegisterWellKnownServiceType(
  typeof(CommonTypes.Library), // type of the remote object
  "Library", // assembly name
  WellKnownObjectMode.SingleCall); // activation mode
```

- **`RegisterWellKnownServiceType(…)`** с параметър указващ че обектът трябва да се използва в **`Singleton`** режим – регистрира тип на отдалечен обект в режим **`Singleton`**. Пример:

```
RemotingConfiguration.RegisterWellKnownServiceType(
  typeof(CommonTypes.Library), // type of the remote object
  "Library", // assembly name
  WellKnownObjectMode.Singleton); // activation mode
```

- **`RegisterActivatedServiceType(…)`** – регистрира отдалечен обект с клиентска активация. Пример:

```
RemotingConfiguration.RegisterActivatedServiceType(
  typeof(CommonTypes.Library)); //type of the remote object
```

Регистрация на клиента (Client-Side Registration)

Чрез програмен код имаме два начина за регистриране на типа на обект, който искаме да ползваме отдалечено от някой Remoting сървър. Използват се методите на класа **RemotingConfiguration:**

- **RegisterWellKnownClientType(…)** – регистрира тип със сървърна активация. Не се указва дали обектът използва режим **SingleCall** или **Singleton**, защото това зависи от сървъра. Клиентът не може и не трябва да знае такива подробности в имплементацията и логиката на сървъра. Форматът на URI адреса, който се подава на метода, е следният:

```
<протокол>://<име на сървъра или IP адрес>:<порт>/<път до
асембли с типа>/<тип>
```

  Ето един пример за регистрация на отдалечен тип с активация от страна на сървъра:

```
RemotingConfiguration.RegisterWellKnownClientType(
  typeof(CommonTypes.Library), // type of the remote object
  "http://remoting_server:1234/remoting/Library"); // object URI
```

- **RegisterActivatedClientType(…)** – регистрира тип с клиентска активация. Форматът на URI идентификаторът е следният:

```
<протокол>://<име на сървъра или IP адрес>:<порт>
```

  Ето пример за регистрация на отдалечен обект, който ползва клиентска активация:

```
RemotingConfiguration.RegisterActivatedClientType(
  typeof(CommonTypes.Library), // type of the remote object
  "http://remoting_server:1234"); // URL of the server
```

Създаване на инстанция на отдалечен обект

За да получим инстанция на даден отдалечен обект трябва преди това да сме регистрирали неговия тип в Remoting инфраструктурата. След това можем да създадем обект от този тип по някой от следните начини:

- Чрез статичния метод **GetObject(…)** на класа **Activator** – използва се за отдалечени обекти със сървърна активация. Създава прокси обект при клиента, чрез който се осъществяват всички извиквания. Комуникация със сървъра се извършва единствено при извикването

на методите и връщането на резултата, а не при създаването на прокси обекта.

```
Library library = Activator.GetObject(
   typeof(Library), // type of the remote object
   "http://remoting_server:1234/remoting/Library"  // object URI
   ) as Library;
```

- Чрез оператора **new** – може да се използва за създаване на обекти със сървърна и с клиентска активация. Осигурява и механизъм за предаване на параметри при инстанциране. В C# операторът **new** се използва както при създаването на локални обекти, така и при създаването на отдалечени обекти. За да създадем отдалечен обект с **new**, преди това съответният клас трябва да е бил регистриран в Remoting инфраструктурата. В противен случай се създава локален обект. Ето примерен код, който създава отдалечен обект с оператора **new**:

```
// This class is located and registered on the server
class Library
{
   private string mName;

   // Default constructor
   public Library()
   {
      mName = "Library";
   }

   // Parametrized constructor
   public Library(string aName)
   {
      mName = aName;
   }
}

...

// This code runs on the client-side

// Register the Library class as remote type
RemotingConfiguration.RegisterWellKnownClientType(
   typeof(CommonTypes.Library),
   "http://remoting_server:1234/remoting/Library");

// Get object from the server using the default constructor
Library library1 = new Library();

//Get object from the server using the parameterized constructor
Library library2 = new Library("My library");
```

Кодът за инстанциране на отдалечен обект с ключовата дума **new** не се отличава от създаването на обикновени обекти. Изобщо работата с отдалечени обекти изглежда в повечето случаи, като че ли се изпълнява върху обикновени инстанции. След регистрация Remoting инфраструктурата се грижи всяко извикване да се прехвърля от локалния прокси клас към отдалечения обект прозрачно, без допълнителна намеса на програмиста.

Маршализация (Marshaling)

Маршализацията (marshaling) е процес на пренос на обекти и данни между Remoting клиента и сървъра. Този процес се извършва при всяко:

- подаване на параметри на метод
- връщане на резултат от метод
- достъп до поле или свойство на обект или тип

Според начина, по който се пренасят има три типа обекти:

- **Marshal-By-Value обекти** – пренасят се по стойност посредством сериализация.
- **Marshal-By-Reference обекти** – пренасят се по референция. По специалното при тях е, че се използва отдалечена референция.
- **Non-Marshaled обекти** – не се пренасят. Причина за това може да бъде, че обектите стават невалидни извън своя application domain или контекст, поради съображения за сигурност и др.

Marshal-By-Value обекти

Marshal-By-Value са всички обекти, които са маркирани с атрибута [**Serializable**] и/или имплементират интерфейса **ISerializable**. Тези обекти се предават по стойност, т.е. тяхно копие се пренася до отдалечената машина. Копират се всички сериализируеми полета и свойства в обекта. Като поведение те са подобни на стойностните типове в .NET Framework. При промяна на данните се променя само локалното копие на обекта. Целият обект се пренася до отдалечената машина наведнъж (само с едно извикваме) и това осигурява добро бързодействие в случай, че обемът на данните не е голям.

Ето един пример, в който са дефинирани два класа, които се маршализират по стойност:

```
[Serializable]
public class Book
{
  private string mAuthor;
  private string mTitle;

  // Default constructor
  public Book()
```

```
   {
   }
   ...
}

public class Author : ISerializable
{
   private string mName;
   private string[] mBooks;

   // Default constructor
   public Author()
   {
   }

   // Constructor for serialization only
   private Author(SerializationInfo info,
      StreamingContext context)
   {
      // Deserialize data to class fields
   }

   // ISerializable.GetObjectData method
   public void GetObjectData(SerializationInfo info,
      StreamingContext context)
   {
      // Serialize class fields
   }
   ...
}
```

Marshal-By-Reference обекти

Marshal-By-Reference са всички обекти, които са наследници на класа **`MarshalByRefObj`** без значение дали типът е сериализируем. Пренасят се до отдалечената машина посредством отдалечена референция, от която се създава динамично прокси клас. Всяко извикване или операция върху обекта се извършва върху прокси класа, който от своя страна прехвърля нужните данни към отдалечената инстанция, където се изпълнява реално извикването. Този процес се повтаря за всяко четене или писане в поле или извикване на метод, което значително понижава производителността.

```
[Serializable]
public class Book : MarshalByRefObject
{
   private string mAuthor;
   private string mTitle;

   // Default constructor
   public Book()
```

```
    {
    }
    ...
}
```

Живот на обектите (Lifetime)

В .NET Framework жизненият цикъл на обектите се управлява от системата за събиране на боклука (**Garbage Collector**). Тя следи дали даден обект се достъпва от клиентите в даден application domain (**AppDomain**). Когато обектът спре да бъде използван той подлежи на събиране от системата за боклук. Ако обектите и клиентите им са в един и същ application domain, системата работи без проблеми. Дори в случай, че клиентът и обектите са в отделни домейни, но в рамките на един и същ процес, отново няма никакви проблеми, тъй като те всичките споделят общ управляван хийп (**managed heap**).

Случаят, в който клиентите и обектите са в различен процес и/или машина, е по-особен. Ако системата за събиране на боклука работи по същия начин, то тя веднага след инициализирането ще открие, че към обекта няма референции и веднага ще го обяви за боклук. За да реши този проблем Remoting инфраструктурата предлага "система за отпускане на "**живот назаем"** за обектите – **lease-based lifetime management**.

Живот назаем

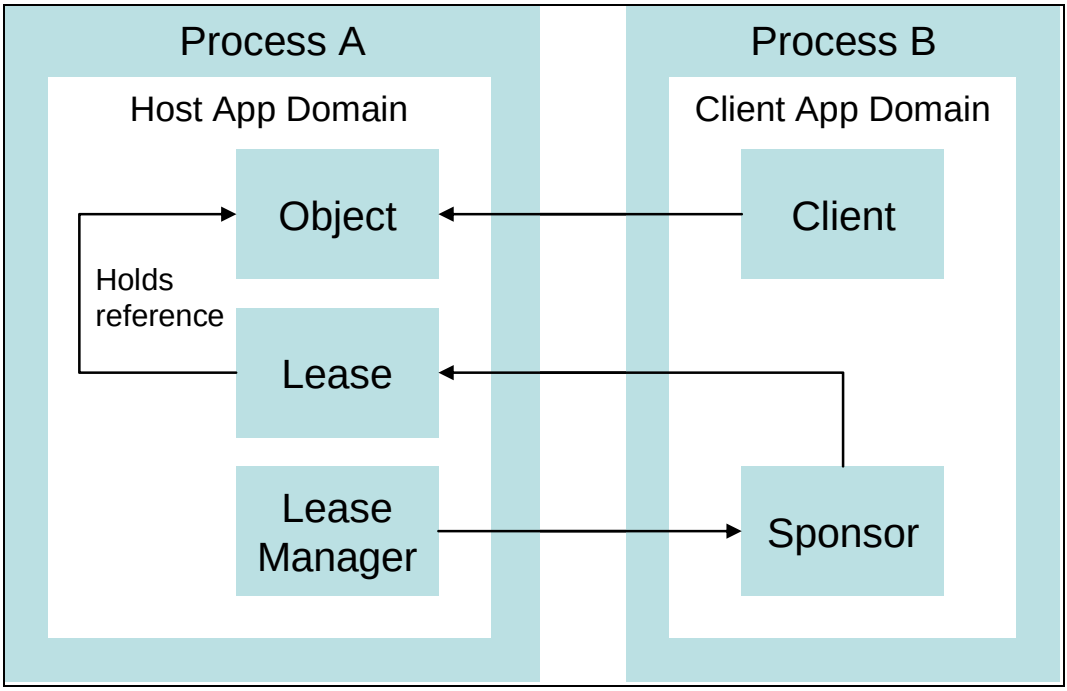
При създаването си всеки отдалечен обект на сървъра се асоциира с **Lease** обект, който има задачата да контролира жизнения му цикъл. Той определя времето, за което отдалечения обект е активен и не подлежи на събиране от системата за системата за почистване на паметта.

Специален обект наречен **Lease Manager** следи сървърните обекти и техните асоциирани **`Lease`** обекти. Всеки Lease обект има начално време, което да дава назаем (initial lease time). Lease времето започва да тече след като първата референция към сървърния обект се маршализира през границата на домейна. Докато то е по-голямо от нула, се счита че обектът, асоцииран с него, все още е активен и CLR смята, че сървърния обект все още се използва от клиента. Lease мениджърът пази референция към обекта, за да предотврати неговото събиране от "системата за почистване на паметта".

Когато Lease времето свърши, CLR решава че обектът не е вече в употреба и че той ресурсите му подлежат на освобождаване. При свършване на времето е възможно да има клиенти, които все още да се нуждаят от обекта. В този момент се отправя заявка към всеки от тях дали все още се нуждаят от обекта.

Спонсори на обектите

Всеки клиент, който иска да бъде "попитан" дали иска да се удължи живота на обекта, който той ползва, трябва да предоставя специален обект-спонсор. Спонсорът от своя страна решава дали да удължи живота на сървърния обект или не. Ако реши да удължи времето за живот, то Lease мениджърът изпълнява желанието му. В противен случай системата за събиране на боклук може да прибере обекта. Следващата схема обяснява нагледно процеса:



Функцията на всеки един от показаните обекти е обяснена по-долу. Тази схема за определяне на продължителността на живота е приложима само за Marshal-By-Reference обекти с клиентска активация и Singleton обекти със сървърна активация. Двата случая има специфични особености, които са разгледани по-долу. Сега ще насочим вниманието си към Lease обекта, спонсора и Lease мениджъра.

Lease обектът имплементира интерфейса **`ILease`**, който предоставя функции за определяне живота на асоциираните с него обекти. Конкретна имплементация на този интерфейс е класът **`Lease`** в **`System.Runtime.Remoting.Lifetime`**.

Интерфейсът ILease

Интерфейсът **`ILease`** се намира в пространството от имена **`System.Runtime.Remoting.Lifetime`**. Както показва схемата, обекти от този тип дават назаем живот на асоциирания си отдалечен сървърен обект. Remoting инфраструктурата автоматично задава стойности по подраз-

биране за свойствата на всяка инстанция от този тип, но е възможно и програмистът да зададе свои собствени. Свойствата, които интерфейсът предоставя са:

- **InitialLeaseTime** – задава времето на живот на обекта. В случай че искаме да зададем друга стойност от тази по подразбиране, трябва да го направим задължително **преди** да сме активирали обекта. По подразбиране времето за живот е 5 минути.
- **RenewOnCallTime** – задава времето, с което е възможно да се увеличи живота на обекта при всяко извикване. По подразбиране стойността му е 2 минути. Важно е да се отбележи, че не е възможно да се получи натрупване на време при интензивно ползване на обекта, тъй като формулата, по която се изчислява новото време за живот, е следната:

```
currentLeaseTime = MAX(InitialLeaseTime -  expiredTime,
  RenewOnCallTime)
```

  С други думи **RenewOnCallTime** стойността ще има ефект, ако е по-голяма от оставащото време за живот, като в такъв случай тя определя оставащо време за живот. Отново сме задължени да зададем тази стойност преди активацията на обекта.

- **SponsorshipTimeout** – задава времето, в което всеки спонсор трябва да отговори на изпратената му заявка за отпускане на още време. В случай, че не бъде получен отговор, се счита, че спонсорът е отказал. Ако нито един от спонсорите не даде допълнително време то обектът подлежи на унищожение. По подразбиране стойността е 2 минути.
- **CurentLeaseTime** – връща колко време още остава от живота на обекта.
- **CurrentState** – връща състоянието на живота на обекта, като стойност от изброения тип **LeaseState**. Стойностите, които връща са: **Initial** в процеса на активация; **Active**, когато оставащото времето за живот е по-голямо от 0; **Renewing** в процеса на добавяне на време при извикване; **Expired** в случай че времето за живот е изтекло; **Null** – при проблем с изчисляването на състоянието. Ето дефиницията на типа **LeaseState**:

```
[Serializable]
public enum LeaseState
{
  Null = 0,
  Initial = 1, // while initializing
  Active = 2, // lease time greater than 0
  Renewing = 3, // while renewing
  Expired = 4 // lease time equal to 0
```

```
}
```

Използване на ILease

В случай, че подразбиращите се стойности не са удобни, можем да променим глобално за цялото приложение стойностите чрез свойствата **LeaseTime**, **RenewOnCallTime** и **SponsorshipTimeout** на класа **LifetimeServices** (в пространството от имена **System.Runtime.Remoting.Lifetime**). Тъй като това са глобални настройки за поведението на отдалечените обекти, те трябва да бъдат направени преди регистрирането на отдалечените типове, които ще използваме. Например:

```
static void Main()
{
   LifetimeServices.LeaseTime = TimeSpan.FromMinutes(10);
   LifetimeServices.RenewOnCallTime = TimeSpan.FromMinutes(5);
   LifetimeServices.SponsorshipTimeout = TimeSpan.FromMinutes(1);

   // Register remotable types or load config file
}
```

Можем да променим стойностите на **Lease** обекта, асоцииран с конкретен отдалечен обект по следния начин:

```
// Type definition
public class ClientActivatedType : MarshalByRefObject
{
}

...

// Create and activate the first instance
ClientActivatedType caLongLiving = new ClientActivatedType();

// Get the Lease object associated with it
ILease longLiving = (ILease)
   RemotingServices.GetLifetimeService(caLongLiving);

// Adjust the lifetime parameters
longLiving.RenewOnCallTime = TimeSpan.FromMinutes(10);
longLiving.SponsorshipTimeout = TimeSpan.FromMinutes(1);

// Create and activate the second instance
ClientActivatedType caShortLiving = new ClientActivatedType();

// Get the Lease object associated with it
ILease shortLiving = (ILease)
   RemotingServices.GetLifetimeService(caShortLiving);
```

```
// Adjust the lifetime parameters
shortLiving.RenewOnCallTime = TimeSpan.FromMinutes(1);

shortLiving.SponsorshipTimeout = TimeSpan.FromSeconds(15);
```

Същите настройки могат да бъдат направени и посредством конфигурационен файл. Структурата и съдържанието на конфигурационния файл ще бъде разгледан в частта "Remoting конфигурационни файлове".

Интерфейсът ISponsor

Интерфейсът **ISponsor** се намира в пространството от имена **System.Runtime.Remoting.Lifetime**. Както споменахме по-рано спонсорът е обект, който има властта да удължава времето на живот на отдалечени обекти. За да може да изпълнява тази роля обектът трябва да имплементира интерфейса **ISponsor**. Единственият метод на този интерфейс е **Renewal(ILease lease)**, който е дефиниран по следния начин:

```
public interface ISponsor
{
   TimeSpan Renewal(ILease lease);
}
```

Lease мениджърът извиква този метод, когато животът на даден обект е изтекъл, за да поиска допълнително време.

За да добавим спонсор към даден отдалечен обект можем да използваме метода на **ILease** обекта **Register(…)**. (За да получим обект от такъв тип трябва да извикаме метода **GetLifetimeService(…)** на желания отдалечен обект, на който ще добавяме спонсор.) Аналогично можем да премахнем даден спонсор от отдалечен обект чрез метода **Unregister(…)** на **ILease**.

```
// Activation of the object
Library library = new Library();

// Getting lease object associated with current object
ILease lease = RemotingServices.GetLifetimeService(library)
   as ILease;

// Creating new sponsor
MySponsor sponsor = new MySponsor();

// Attaching sponsor to our lease object
lease.Register(sponsor);

// Detaching sponsor from our lease object
lease.Unregister(sponsor);
```

Премахването на спонсора, посредством метода **Unregister(…)**, от списъка със спонсори на отдалечения обект не е задължително, но на прак-

тика, ако не се използва, води до голяма загуба на производителност, тъй като Remoting системата отправя заявки и чака отговор от **всеки** регистриран спонсор. Така че препоръчително е след като даден спонсор стане ненужен, да бъде изваден от списъка на валидните спонсори.

Важна особеност на класовете, които играят ролята на спонсори е, че те реално се извикват през границите на домейна на приложението. Поради това спонсорът трябва да бъде отдалечен обект, който се маршализира по стойност или по референция.

Спонсор, маршализиран по стойност

При спонсори, които са маркирани единствено като сериализируеми (с атрибута `[Serializable]`), при регистриране на спонсор той се маршализира по стойност до сървърната страна. От там нататък сървърът използва своето копие на спонсора. Това осигурява по-добро бързодействие тъй като спестява маршализирането на спонсора за всяко извикване. Този начин ни дава възможност да контролираме живота на обектите от гледна точка на натоварването на сървъра (което не винаги е показателно за това, дали обектът все още е нужен на клиента!). За съжаление при решаването дали да удължи живота на даден обект спонсорът може да използва само информацията, която има на сървъра, тъй като е отделен от клиентската част.

Спонсор, маршализиран по референция

В случай, че спонсорът наследява `MarshalByRefObject`, той се намира при клиента. Тъй като има достъп до клиентската част, той може да базира своите решения на информацията, която получава от клиента, като следи определени негови събития или свойства.

В този случай възниква следният въпрос: ако Lease обектът пази "жив" отдалечения сървърен обект, и ако спонсорът пази Lease обекта "жив", какво пази спонсора "жив"? Отговорът е, че при клиента трябва да се държи референция към спонсора и обикновено това се реализира чрез член-променлива на някой подходящ клас. По този начин клиентът има възможност да дерегистрира своите спонсори когато завършва изпълнението си. Това може да се осъществи и в метода `Dispose()` ако класът имплементира интерфейса `IDisposable`. Както вече споменахме дерегистрирането на спонсора подобрява значително производителността, тъй като Lease мениджърът не губи време да достъпва невалиден спонсор.

Друга характерна особеност на този тип спонсори е, че те са callback обекти от гледна точка на сървъра. Поради това и поради изисквания към сигурността в .NET за да работят този вид спонсори трябва да укажем при конфигурирането на каналите следните свойства:

- Клиентът трябва да регистрира порт за всеки канал. Това е нужно за да може Lease мениджърът да може да извиква спонсора. По принцип няма значение точно кой порт ще бъде подаден. Добра практика е да се подава порт 0, тъй като в този случай Remoting

системата сама избира някой порт. Каналът, номерът на порта и местонахождението на спонсора се изчисляват когато се маршализира референцията на спонсора до сървъра.

Живот на Singleton отдалечени обекти

Семантиката на Singleton обектите изисква те да имат неограничен живот. Посредством стандартните процедури за определяне на продължителността на живота в Remoting системата, това е невъзможно или най-малкото неудобно и неефективно. За да се реши този проблем при проектирането на Singleton обекта трябва да се предефинира метода **`InitializeLifetileService()`** на базовия клас **`MarshalByRefObject`** по следния начин:

```
public class MySingleton : MarshalByRefObject
{
   public override object InitializeLifetimeService()
   {
      // Returning null as Lease object
      // indicates that lease never expires
      return null;
   }
}
```

Връщайки **`null`** като резултат от метода ние указваме, че искаме този обект да има безкраен живот. По този начин също решаваме проблема със спонсорите и Lease свойствата определящи продължителността на живота на обекта.

Remoting конфигурационни файлове

В настоящата секция ще разгледаме конфигурационните файлове на Remoting инфраструктурата и ще дадем описание на таговете, които се използват в тях. Тъй като сървърът и клиентът имат свои специфични елементи ще ги разгледаме последователно.

Remoting конфигурационните файлове представляват XML файлове със специална структура. Работата с тях е лесна и най-важното – те предоставят гъвкав начин за конфигуриране на приложения без да се налага прекомпилиране (за разлика от варианта, в който настройките са зададени в програмния код). В практиката е силно препоръчително да конфигурацията да се държи в XML а не в програмния код.

За да накараме Remoting системата да използва този файл трябва да извикаме статичния метод **`RemotingConfiguration.Configure(…)`** с единствен параметър – пътят и името на конфигурационния файл:

```
RemotingConfiguration.Configure("remoting.config");
```

Пътят до файла, името и разширението му нямат значение, освен в случай на хостинг на асемблитата със споделените класове в IIS. Remoting конфигурационните файлове са допълнителни и са отделни от конфигурационните файлове на приложението.

След като сме извикали този метод можем да пристъпим към активиране и използване на отдалечените обекти без никакви по-нататъшни грижи за канали и форматери.

Структура и елементи на конфигурационния файл

В тази част от главата ще разгледаме подробно повечето от елементите, които могат да се съдържат в конфигурационния файл на приложение, използващо Remoting. Освен наличието на предефинираните XML тагове, в определени случаи е важно и тяхното разположение спрямо останалите тагове в конфигурацията. Ще започнем разглеждането от корена на XML документа – това е тагът **<configuration>**.

<configuration>

Този таг е коренът на всички елементи във файла. Трябва да се среща точно веднъж.

<system.runtime.remoting>

Намира се задължително като под-елемент на **<configuration>**. В този таг трябва да се намират всички елементи свързани с конфигурирането на Remoting инфраструктурата. Може да се среща само веднъж.

<application>

Представлява задължително под-елемент на **<system.runtime.remoting>**. Съдържа всички специфични за приложението данни. Може да се среща само веднъж.

Има незадължителен атрибут **name**, който указва името на приложението:

```
<application name="RemotingApp">
   ...
</application>
```

<lifetime>

Намира се винаги под елемента **<application>**. Служи за конфигуриране на времето на живот на обектите. Важи за всички обекти на това приложение. Тези настройки се отнасят за обектите с клиентска активация и за **Singleton** обектите. Те имат ефект само в конфигурационния файл на сървъра, т.е. ако в конфигурационния файл на клиента има такъв таг, той бива игнориран. Конфигурирането на времената, свързани с жизнения цикъл, става чрез атрибутите на тага:

- **leaseTime** – времето "на заем" на всеки обект, свързан с даденото приложение. По подразбиране има стойност 5 минути.
- **sponsorshipTimeout** – времето, което Lease мениджърът изчаква отговора на спонсора след като го уведоми че даден Lease е изтекъл. Ако спонсорът не отговори в този период обектът подлежи на освобождаване от системата за събиране на боклука. По подразбиране времето е 2 минути.
- **renewOnCallTime** – времето, с което се увеличава продължителността на живота на обект, при всяко негово извикване. За повече подробности относно начина на изчисляване на допълнителното време вижте частта "Интерфейсът **ILease**". Подразбиращата се стойност е 2 минути.
- **leaseManagerPollTime** – времето, което Lease мениджърът изчаква, след като е проверил системата за Lease обекти с изтекло време, преди да започне нова проверка. По подразбиране периодът е 10 секунди.

Единиците за измерване на времето се записва веднага след стойността за всеки атрибут. Символите за мерните единици не са зависими от главни и малки букви. Валидните стойности са:

- **D** – дни
- **H** - часове
- **M** - минути
- **S** - секунди
- **MS** – милисекунди

Ето как изглежда **<lifetime>** секцията в един примерен конфигурационен файл:

```
<application>
  <lifetime
     leaseTime="3m"
     sponsorshipTimeout="30s"
     renewOnCallTime="1m"
     leaseManagerPollTime="750ms" />
  ...
</application>
```

<service>

Представлява под-елемент на **<application>**. Не е задължителен елемент и може да се среща повече от веднъж в рамките на един конфигурационен файл. Използва се като контейнер, в който са изброени и описани всички отдалечени типове, които сървърът предоставя за използване. Поради това има смисъл само в приложения, които играят ролята на

сървър. Тъй като дадено приложение може да бъде едновременно както клиент така и сървър, този таг е съвместим с тага **<client>**. В този таг се съдържат таговете **<wellknown>** и **<activated>**, които са разгледани по-долу.

<client>

Представлява под-елемент на **<application>**. Не е задължителен елемент и може да се среща повече от веднъж. Служи за контейнер, в който са изброени и описани типовете на отдалечените обекти, които приложението може да използва. Може да съдържа таговете **<wellknown>** и **<activated>**. Има следните атрибути:

- **url** – определя URL адреса, който се използва за активиране на клиентски обекти. Ако приложението използва такъв тип обекти, този атрибут е задължителен.
- **displayName** – използва се само от приложението Admin Tool, за да може потребителите му да отличават визуално различните такива тагове, когато са използвани повече от един. Този атрибут е незадължителен.

<wellknown>

Може да се използва като под-елемент само на таговете **<service>** и **<client>**. И в двата случая този таг е незадължителен и може да се среща повече от веднъж. Тъй като между начините, по които се използва в двата случая, има различия, ще ги разгледаме поотделно.

В конфигурационния файл на сървъра този таг се намира под тага **<service>**. Той описва отдалечените типове със сървърна активация, предлагани от сървъра. Чрез него се описват **SingleCall** и **Singleton** обектите. Има следните задължителни атрибути:

- **mode** – типа на активирания на сървъра обект. Валидните стойности са **SingleCall** и **Singleton**.
- **type** – означава пълния тип на обекта. Има следния формат:

<пълен тип на обекта>, **<асембли, в което се намира типът>**, Version=**<версия>**, PublicKeyToken=**<публичен ключ на силно именуваното асембли>**, Culture=**<култура>**

Version, **PublicKeyToken** и **Culture** се използват само при силно именувани асемблита.

- **objectUri –** URI адрес на отдалечения обект, към който клиентът се обръща. Трябва да бъде уникално за всеки тип в приложението. Зависим е от малки и главни букви. Особеност при този атрибут е, че когато обектът се достъпва през IIS (Internet Information Services), е нужно той да завърша с разширение **.soap**. Повече за използването

на Remoting чрез IIS можете да намерите в частта "Remoting сценарии".

В конфигурационния файл на клиента **<wellknown>** описва типовете със сървърна активация, които клиентът използва. Атрибутите, които се използват в този случай са:

- **type** – означава пълния тип на обекта. Има същия формат като случая със сървърната конфигурация.
- **url** – адресът, който трябва да се използва, за да може клиентът да се свърже със сървъра и неговите типове. Съдържанието на този адрес зависи и от това дали в **<application>** тага е въведено име на приложението.

За да представим нагледно конфигурационните възможности нека разгледаме следващите два примера. Конфигурация на Remoting сървър:

server.config

```
<service>
  <wellknown mode="SingleCall"
       type="CommonTypes.Query, CommonTypes"
       objectUri="Query" />
  <wellknown mode="Singleton"
       type="CommonTypes.Library, CommonTypes"
       objectUri="Library" />
</service>
```

Конфигурация на Remoting клиент:

client.config

```
<client>
  <wellknown type="CommonTypes.Query, CommonTypes"
    url="http://remoting_server:1234/Query" />
  <wellknown type="CommonTypes.Library, CommonTypes"
    url="http://remoting_server:1234/Library" />
</client>
```

<activated>

Представлява под-елемент на таговете **<service>** и **<client>** съответно в сървърния и в клиентския конфигурационен файл. Описва типовете с клиентска активация, които приложението предлага или използва. Не е задължителен елемент и може да се среща многократно в тези две секции. Има един задължителен атрибут **type**, който се използва и в двата случая за описване на пълния тип на отдалечения обект. Ето пример за конфигуриране на клиентска активация от страна на сървъра:

server.config

```
<service>
  <activated type="CommonTypes.Book, CommonTypes" />
</service>
```

Ето как изглежда съответната конфигурация от страна на клиента:

client.config

```
<client>
  <activated type="CommonTypes.Book, CommonTypes" />
</client>
```

<channels>

Представлява под-елемент на тага **<system.runtime.remoting>** или **<application>**.

Когато се намира под **<system.runtime.remoting>**, с този таг се дефинират нови канали, създадени от разработчика.

В случая, когато той се намира под **<application>**, конфигурираме вече съществуващи или вече описани канали в секцията **<channels>** под **<system.runtime.remoting>**.

Стандартните канали като TCP и HTTP каналите са описани във файла **machine.config**. Ако искаме да променим тяхното поведение глобално, за цялата машина, можем да редактираме този файл, но това не е препоръчителна практика, тъй като може да отворим дупка в сигурността на машината. Друг проблем е, че ако нашето приложение изисква такива промени, то трудно би могло да се разпространява, тъй като трябва да се налага административна намеса в процеса на инсталиране, а също така е възможно да се получат несъвместимости с други приложения, които разчитат на стандартните настройки. Затова е по-добре да се използват локално дефинирани канали.

<channel>

Може да се използва като под-елемент на **<channels>** без значение къде се намира родителската секция. С този таг се конфигурират параметрите на всеки един от каналите, които приложението използва. Може да се съдържа в една **<channels>** секция повече от веднъж. Поради това, че в зависимост от разположението на родителския таг, тази секция има различно значение, ще разгледаме двата случая по отделно.

Когато **<channels>** се намира под **<system.runtime.remoting>** можем да разглеждаме тази секция като шаблон на канала. Тагът **<channel>** описва канала и някои от неговите свойства като например:

- **id** – уникално име на канала, което се използва за реферирane на канала в други части на конфигурационния файл. Не трябва да се допускат канали с еднакви **id** атрибути, тъй като парсерът не предупреждава за грешка, а използва навсякъде последно декларирания канал с това **id**. Този атрибут е задължителен.
- **type** – пълно име на класа, който имплементира канала. Описва се в същия формат, който се използва при атрибута **type** на тага **<wellknown>**. Задължителен атрибут.
- **name** – име на канала. Използва се, когато се налага да се регистрира един и същ канал, който да "слуша" на повече от един порт.
- **priority** – приоритет на канала спрямо другите регистрирани от приложението канали. При заявка към сървъра Remoting системата използва тази информация, за да намери подходящ канал за връзка. По-голям приоритет имат каналите с по-голяма стойност на този атрибут. Стандартните канали, които са част от .NET Framework, имат приоритет равен на 1. Отрицателните числа са също валидни стойности.
- **displayName** – използва се само от приложението Admin Tool, за да може потребителите му да отличават визуално различните такива тагове, когато са използвани повече от един. Този атрибут е незадължителен.
- Специфични за канала свойства. Тъй като свойствата на каналите не се контролират от Remoting инфраструктурата и всеки канал може да има специфични изисквания, чрез тази специална група от атрибути може да се подават специфичните стойности. За тях няма предефинирани атрибути. Форматът, в който се подават стойностите, е ключ-стойност. За по-нагледно представяне нека разгледаме един пример:

```
<channel id="customChannel"
   type="CommonTypes.Channels.CustomChannel, CommonTypes"
   myProperty="myValue"
   author="Viktor Zhivkov" />
```

Когато използваме **<channels>** в рамките на **<application>** тага ние се обръщаме към вече дефиниран канал в друга **<channels>** секция или в **machine.config** файла. В този случай трябва да укажем следните задължителни атрибути:

- **ref** – означава името (**id** атрибута) на шаблона, който реферираме.
- **port** – номера на порта, на който каналът трябва да "слуша". В клиентските конфигурационни файлове можем да зададем стойност 0, при което Remoting системата автоматично избира вместо нас подходящия порт за връзка.

Каналите имат и други атрибути освен изброените задължителни. Ще разгледаме поотделно списъка с атрибути на стандартните вградени HTTP и TCP канали.

Атрибути на HTTP канал:

- **clientConnectionLimit** – определя максималния брой на едновременните връзки за даден канал. По подразбиране има стойност 2.
- **proxyName** – име на прокси сървър. По този начин се конфигурира Remoting инфраструктурата да използва прокси сървър.
- **proxyPort** – порт на прокси сървъра, който да се използва за комуникация. Употребява се заедно с **proxyName** атрибута.
- **useIpAddress** – булева стойност за това дали в URL адресите на предоставяните типове се използва IP адрес (**true**) или име на машина (**false**). Приложим е само в конфигурацията на сървъра. По подразбиране има стойност **true**.
- **machineName** – име на машината, което да се използва при комуникация вместо истинското й име. Ако е подадена стойност автоматично атрибутът **useIpAddress** приема стойност **false**.

Атрибути на TCP канал:

- **useIpAddress** – булева стойност за това дали в URL адресите на предоставяните типове се използва IP адрес (**true**) или име на машина (**false**). Приложим е само в конфигурацията на сървъра. По подразбиране има стойност **true**.
- **rejectRemoteRequests** – булева стойност, която задава дали да се отхвърлят връзки от други машини. Когато има стойност **true** са разрешени само извиквания между процесите на една машина.

<channelSinkProviders>

Намира се под тага **<system.runtime.remoting>** и служи за контейнер на таговете, описващи тръбите на канала (**channel sink**). Тази секция от конфигурационния файл не е задължителна и може да се среща най-много веднъж във файл. Валидните под-елементи на този таг са **<serverProviders>** и **<clientProviders>**

<serverProviders>

Този елемент може да бъде използван както под тага **<channelSinkProviders>**, така и под **<channel>**. Той служи за дефиниране и конфигуриране на доставчиците (providers) от страна на сървъра.

По аналогия с каналите първата употреба дефинира нов sink provider, а втората конфигурира вече съществуващ. Стандартните доставчици са дефинирани във файла **machine.config**. Този таг може да се употребява само веднъж.

Стандартните доставчици и форматери на канала се преконфигурират, ако използваме този таг за реферирание или деклариране в **<channel>** секцията. В този случай трябва да изброим всички доставчици и форматери, които ще бъдат използвани от канала. Например, ако добавим в конфигурационния файл на клиента доставчик, а след това форматер, ще получим изключение (exception), ако доставчикът не имплементира нужен на форматера интерфейс.

В тази секция могат да се използват таговете **<formatter>** и **<provider>**.

<clientProviders>

Тагът **<clientProviders>** е аналогичен на **<serverProviders>**, с тази разлика, че дефинира и конфигурира доставчиците от страна на клиента.

<provider>

Използва се в секциите **<serverProviders>** и **<clientProviders>**. Описва sink provider на канала, за който се отнася. Може да се среща нула или повече пъти в една секция. За конфигуриране се използват следните задължителни атрибути:

- **id** – уникално име на доставчика, което ще се използва за реферирание.
- **type** – пълен тип на класа, чиято инстанция ще бъде доставчика. Форматът на този атрибут е същият, както на досега разглежданите атрибути **<type>**.
- **ref** – посочва **id** на доставчика, който се реферира. Не може да се използва в секции, които дефинират такива.

Освен тези атрибути можем да подадем на конструктора на доставчика параметри чрез съдържанието на **<provider>** тага. От гледна точка на Remoting системата, няма значение какво е името на тага вътре, тъй като той се подава като DOM структура. Всички XML атрибути и XML структури в този таг се подават на конструктора на доставчика, описан в конфигурацията. Всеки един от доставчиците трябва да има конструктори, които приемат **IDictionary** или **ICollection** като входни параметри. Речниковата колекция се използва за контейнер на подадените атрибути, а колекцията – за **DOM** структурата на елемента и неговите под-елементи.

Ето така изглежда част от един примерен конфигурационен файл. Тагът **<filter>**, както и атрибутите **mode**, **mask**, **ip** ще бъдат използвани като параметри за инстанцирането на доставчика от тип **IpFilter** (този тип е измислен за примера, не го търсете в .NET Framework!).

```
...
<system.runtime.remoting>
  <channels>
    <channel ref="tcp" port="1234">
```

```
        <serverProviders>
          <provider ref=ipFilter" mode="accept">
            <filter mask="255.255.255.255" ip="192.168.1.1" />
          </provider>
        </serverProviders>
      </channel>
    </channels>
...
    <channelSinkProviders>
      <serverProvider>
        <provider id="ipFilter"
    type="CommonTypes.Providers.IpFilter, CommonTypes" />
      </serverProvider>
    </channelSinkProviders>
</system.runtime.remoting>
```

<formatter>

Използва се като под-елемент на **<serverProviders>** и **<clientProviders>**. Конфигурира какъв форматер ще използва даденият канал. Ако бъде пропуснат, се използва подразбиращият се форматер. Може да се изброят няколко форматера, които да обработват последователно данните, преминаващи през канала. В този случай другата страна в комуникацията трябва да бъде конфигурирана със същата последователност от форматери. За конфигуриране на форматера се използват следните атрибути:

- **id** – уникално име на форматера, което ще се използва при реферирane.
- **type** – пълен тип на класа, чиято инстанция ще бъде форматер. Форматът на този атрибут е същия, както в досега разглежданите атрибути **<type>**.
- **ref** – посочва **id** на форматера, който се реферира. Не може да се използва в секции, които дефинират такива.
- **typeFilterLevel** – определя нивото на позволените извиквания. Има две стойности: **Low** и **Full**. Подразбиращата се стойност **Low** не позволява обръщения от тип callback. В случай, че ни се налага да използваме такива, трябва да променим тази стойност на **Full**. Най-честата причина за промяна на тази настройка на форматера е използването на спонсори с маршализация по референция (те имат нужда от callback извиквания).

Подобно на доставчиците, и форматерите могат да бъдат фино конфигурирани с потребителски атрибути и тагове в **<formatter>** секцията. Процесът е аналогичен на разгледания по-горе.

Два реални конфигурационни файла – пример

За да обобщим и придадем форма на натрупаните до тук факти за конфигурационните файлове, нека разгледаме два цялостни реални примера:

Пример за цялостен конфигурационен файл за Remoting инфраструктурата от страна на сървъра:

server.config

```
<configuration>
  <system.runtime.remoting>
    <application name="RemotingApp">
      <service>
        <wellknown mode="SingleCall"
             type="CommonTypes.Query, CommonTypes"
             objectUri="Query" />
        <wellknown mode="Singleton"
             type="CommonTypes.Library, CommonTypes"
             objectUri="Library" />
        <activated type="CommonTypes.Book, Book" />
      </service>
      <channels>
        <channel ref="tcp" port="1234">
          <serverProviders>
            <formatter ref="binary" typeFilterLevel="Full" />
          </serverProviders>
          <clientProviders>
            <formatter ref="binary"/>
          </clientProviders>
        </channel>
        <channel ref="http" port="1235">
          <serverProviders>
            <formatter ref="soap" typeFilterLevel="Full" />
          </serverProviders>
          <clientProviders>
            <formatter ref="soap"/>
          </clientProviders>
        </channel>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Пример за цялостен конфигурационен файл за Remoting инфраструктурата от страна на клиента:

client.config

```
<configuration>
  <system.runtime.remoting>
```

```
    <application name="RemotingApp">
      <lifetime
        leaseTime="3m"
        sponsorshipTimeout="30s"
        renewOnCallTime="1m"
        leaseManagerPollTime="750ms" />
      <client url="http://remoting_server">
        <wellknown type="CommonTypes.Query, CommonTypes"
            url="http://remoting_server/RemotingApp/Query" />
        <wellknown type="CommonTypes.Library, CommonTypes"
            url="http://remoting_server/RemotingApp/Library"/>
        <activated type="CommonTypes.Book, Book" />
      </client>
      <channels>
        <channel ref="tcp" port="0">
          <serverProviders>
            <formatter ref="binary"/>
          </serverProviders>
          <clientProviders>
            <formatter ref="binary"/>
          </clientProviders>
        </channel>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

В тези два файла са показани най-важните и често срещани елементи. В случай, че използвате само вградените канали, доставчици и форматери, ще са ви достатъчни само такива конфигурационни файлове.

Remoting сценарии

Освен Remoting системата, .NET Framework предоставя и други начини за взаимодействие между обекти в различни домейни на приложението (application domains). Всеки от тях е създаден с определена цел, гъвкавост и изисквания към програмистите, които го ползват.

Най-големият конкурент на Remoting технологията са уеб услугите. Поради това, че те използват HTTP протокола и SOAP сериализация, те много приличат на Remoting решение, използващо HTTP канал и SOAP форматер. От своя страна Remoting технологията ни позволява да поставим асемблитата, които съдържат типовете, които ще използваме като отдалечени в IIS. Приликите между двете технологии не се изчерпват само с разгледаното до тук. Възниква въпросът, в кои случаи кое решение е по-удачно.

При избор на технология за конкретна ситуация, трябва преценим доколко сме опитни във всяка една от тях и доколко тя ще бъде удобна

за работа. Нека разгледаме критерии, по които да се ръководим при вземането на такова решение, по ред на техния приоритет.

1. Изисквания към сигурността: когато трябва да криптираме данните и извикванията и/или да автентикираме потребителите, е добре да използваме HTTP базирано приложение, което се намира в IIS. По този начин използваме средствата, които IIS предлага и намаляваме част от отговорността и натоварването от себе си. Имаме свободата да използваме както уеб услуги така и Remoting. Ако решим да използваме Remoting извън IIS, то трябва сами да се погрижим да защитим своите данни и код.

2. Производителност: като цяло Remoting технологията е по-бърза и по-производителна от своите конкуренти. Най-добрата комбинация за случая е TCP канал с бинарна сериализация. Уеб услугите биха били по-добър избор, когато нямаме нужда от характерните за Remoting възможности, а сме задължени да използваме HTTP канал със SOAP сериализация.

3. Съвместимост: при такива изисквания изборът ни са уеб услугите. Remoting технологията е оптимизирана за работа с .NET клиенти. За да се постигне по-добра съвместимост с други технологии (Java, PHP, C++ и др.) трябва да се използва SOAP форматиране на съобщенията, което накланя везните в полза на уеб услугите. Въпреки това можем да поставим нашето Remoting приложение в IIS, да използваме HTTP канал и SOAP форматер и да се възползваме от сигурността и мащабируемостта, която той предлага.

4. Мащабируемост (scalability): в този случай единствената препоръка е да се използва IIS като среда за изпълнение, независимо дали сме се спрели на Remoting или уеб услуги.

5. Функционалност на CLR: чрез Remoting можем да използваме по-пълно възможностите на .NET Framework. Някои от тях не са на разположение при уеб услугите, като например:
   - интерфейси
   - контекст на извикването
   - свойства
   - индексатори
   - управлявани разширения за C++
   - идеално съответствие между всички типове използвани от клиента и сървъра
   - делегати

6. Обектно-ориентиран дизайн на приложението: XML уеб услугите не отговарят напълно на обектно-ориентирания дизайн. Като цяло те са уеб ресурси, които подобно на уеб страниците стандартно не поддържат състояние. За разлика от тях Remoting обектите са обекти в пълния

смисъл на думата. Като резултат тази технология има следните обектно-ориентирани възможности, които отсъстват уеб услугите:

- обектни референции към отдалечени обекти
- няколко възможности за активиране на обект
- обектно-ориентирано управление на състоянието
- разпределено управление на живота на обектите

Това са ключовите точки, които трябва да се обмислят при избора на технология за реализация на приложение, което използва отдалечени обекти. Като обобщение на всичко до тук, нека разгледаме поотделно всяка една от наличните технологии за тази цел.

Чиста мрежова комуникация

Използвайки класовете от пространството с имена **`System.Net`** можем да изградим от нулата своя собствена система за комуникация. Можем да имплементираме свои собствени канали, форматери, протоколи и т.н. Проблемът с този подход е, че се работи на ниско ниво и се хвърлят много усилия за "преоткриване на топлата вода".

XML уеб услуги

Ако ще разработваме уеб приложение и разполагаме с възможностите на ASP.NET, то XML уеб услугите са почти винаги правилният избор. Ползването на отворените стандарти XML и SOAP ги прави изключително съвместими, но в някои случаи това е минус, тъй като няма идеално съответствие между типовете, които клиентите използват, за да извлекат данните.

.NET Remoting

Тази система за комуникация е гъвкава, разширяема и не изисква писането на много код. Можем да я използваме и по начин, подобен на уеб услугите. Преимуществата, които тя предлага са:

- възможност за публикуване и използване на сървърни обекти, от който и да е тип в произволен application domain (конзолно приложение, Windows или уеб приложение, уеб услуга).
- запазване на съответствието на типовете при бинарна сериализация
- възможност за предаване на обекти по референция
- контрол над процеса на активация и живот на обектите
- възможност да използваме разработени от трети страни канали и протоколи, за да разширим начините за комуникация
- възможност за директно участие в процеса на комуникация и да го управляваме според нашите нужди

Remoting сървър и клиент – пример

Време е да разгледаме на практика как изглежда едно приложение, което използва Remoting. Ще реализираме просто конзолно приложение от тип клиент/сървър, което ще обслужва библиотека с книги. Приложението се състои от две части – Remoting сървър и Remoting клиент.

Кодът на приложението е разпределен в три проекта в едно VS.NET решение (solution):

- **CommonTypes** – в този проект се намират общите типове, които клиентът и сървърът ще използват.
- **LibraryServer** – сървърното приложение, което ще ни осигурява достъп до отдалечените обекти.
- **LibraryClient** – клиентското приложение, което осъществява достъп до отдалечената библиотека.

Създаване на общите типове

За да реализираме нужната функционалност се нуждаем от следните типове:

- **Book** – представлява една книга. Има три частни полета – име, автор и ISBN. Достъпът до тях се осъществява чрез публични свойства. За да можем да следим изпълнението на приложението при извикването на тези свойства се отпечатва съобщение в конзолата на приложението. Класът наследява **MarhalByRefObject**, поради което на клиента се предоставя отдалечена референция и всички извиквания се извършват на сървъра. Класът изглежда по следния начин:

Book.cs

```
using System;

namespace CommonTypes
{
  public class Book : MarshalByRefObject
  {
    private string mAuthor;
    private string mTitle;
    private string mIsbn;

    public string Author
    {
      get
      {
        Console.WriteLine("Book's author retrieved.");
        return mAuthor;
      }
```

```
        set
        {
          mAuthor = value;
          Console.WriteLine("Book's author updated.");
        }
      }

      public string Title
      {
        get
        {
          Console.WriteLine("Book's title retrieved.");
          return mTitle;
        }

        set
        {
          mTitle = value;
          Console.WriteLine("Book's title updated.");
        }
      }

      public string Isbn
      {
        get
        {
          Console.WriteLine("Book's ISBN retrieved.");
          return mIsbn;
        }

        set
        {
          Console.WriteLine("Book's ISBN updated.");
          mIsbn = value;
        }
      }

      public Book(string aAuthor, string aTitle, string aIsbn)
      {
        mAuthor = aAuthor;
        mTitle = aTitle;
        mIsbn = aIsbn;
      }
    }
}
```

- **Library** – класът представлява библиотека от книги. Съдържа масив от всички книги в библиотеката и публичен метод, който осигурява тяхното извличане. В приложението трябва да има една единствена

инстанция на този клас, т.е. **Library** е **Singleton** обект. Класът също наследява **MarhalByRefObject** и се маршализира по референция. За да осигурим безкраен живот на обекта, предефинираме метода **InitializeLifetimeService()** на базовия клас, като му указваме да връща винаги **null**. Класът изглежда така:

Library.cs

```
using System;

namespace CommonTypes
{
   public class Library : MarshalByRefObject
   {
      private Book[] mBooks;

      public Library()
      {
         Console.WriteLine("Library object activated.");
         mBooks = new Book[]
         {
            new Book("Steve McConnell", "Code Complete 2",
               "0735619670"),
            new Book("Svetlin Nakov",
               "Internet Programming in Java", "9547753053"),
            new Book("Martin Fowler", "Refactoring: Improving " +
               "the Design of Existing Code", "0201485672")
         };
      }

      public Book[] GetBooks()
      {
         Console.WriteLine("Library.GetBooks() called.");
         return mBooks;
      }

      public override object InitializeLifetimeService()
      {
         return null;
      }
   }
}
```

Създаване на сървър

Имплементацията на сървъра не е сложна. Той представлява конзолно приложение, чиято отговорност е да предоставя отдалечени обекти на клиентите.

Основните стъпки при неговата реализация са следните:

1. Регистриране на канал – регистрираме TCP канал на порт 12345 с подразбиращия се бинарен форматер.
2. Регистриране на типовете отдалечени обекти – регистрираме типа **`CommonTypes.Library`** като **`Singleton`** обект със сървърна активация.
3. Замразяваме изпълнението на сървърното приложение, за да предотвратим неговото завършване, тъй като ако сървърът завърши своето изпълнение, отдалечените обекти стават недостъпни. Не трябва да ни обърква фактът, че сме "приспали" главната нишка на самото приложение, защото неговата единствена цел е да регистрира каналите и типовете, които клиентите ще използват. Всички останали манипулации, свързани с комуникацията между отдалечените обекти и клиента, се извършват автоматично от Remoting инфраструктурата в нишки, различни от главната.

Кодът на сървърното приложение е следният:

LibraryServer.cs

```
using System;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

using CommonTypes;

namespace LibraryServer
{
   class LibraryServer
   {
      const int LISTENING_PORT = 12345;

      static void Main()
      {
         // Create the Remoting TCP channel and register it
         TcpChannel channel = new TcpChannel(LISTENING_PORT);
         ChannelServices.RegisterChannel(channel);

         // Register the Library class as singleton server
         // activated object
         RemotingConfiguration.RegisterWellKnownServiceType(
            typeof(CommonTypes.Library), "Library",
            WellKnownObjectMode.Singleton);

         Console.WriteLine("Library remoting server is " +
            "listening on TCP port {0}", LISTENING_PORT);
         Console.WriteLine("Press [Enter] to exit.");
         Console.ReadLine();
      }
   }
}
```

Създаване на клиент

Клиентското приложение е аналогично на сървърното. Отново за простота използваме конзолно приложение, което ще има за цел да извлече данните за всички книги от библиотеката на сървъра, после да промени автора на една от тях и да отпечата данните за всички книги в конзолата. Отново минаваме през стандартните стъпки при работа с Remoting:

1. Регистриране на канал – регистрираме TCP канал на порт 12345 със подразбиращия се бинарен форматер. Трябва каналите от двете страни на комуникацията да са едни и същи, за да може тя да се осъществи успешно.
2. Активиране на отдалечен обект – клиентът получава референция към единствената инстанция, която е на сървъра, и от тук нататък може да работи с нея сякаш е локална за приложението.
3. Използваме отдалечения обект, както локален – в случая извикваме метода **`GetBooks()`** на класа **`Library`** и след това осъществяваме достъп до свойствата на класа **`Book`**.

LibraryClient.cs

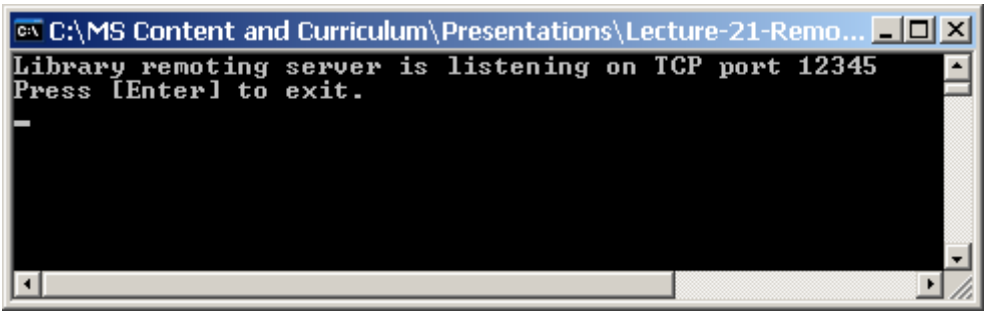
```
using System;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

using CommonTypes;

namespace LibraryClient
{
  class LibraryClient
  {
    public static void Main()
    {
      // Create new client TCP channel and register it
      TcpChannel channel = new TcpChannel();
      ChannelServices.RegisterChannel(channel);
      Console.WriteLine("Registered new client TCP channel.");

      // Activate the Library remote object
      Library remoteLibrary = (Library)
        Activator.GetObject(typeof(Library),
        "tcp://localhost:12345/Library");
      Console.WriteLine("The Library object activated.");

      // Retrieve the books from the server
      Book[] books = remoteLibrary.GetBooks();

      // Update the first book (through a server call)
      books[0].Author = "Author changed";
```

```
        // Print books (through a series of server calls)
        foreach (Book book in books)
        {
           Console.WriteLine("(Author: {0}; Title: {1}, ISBN:
              {2})", book.Author, book.Title, book.Isbn);
        }
     }
  }
}
```

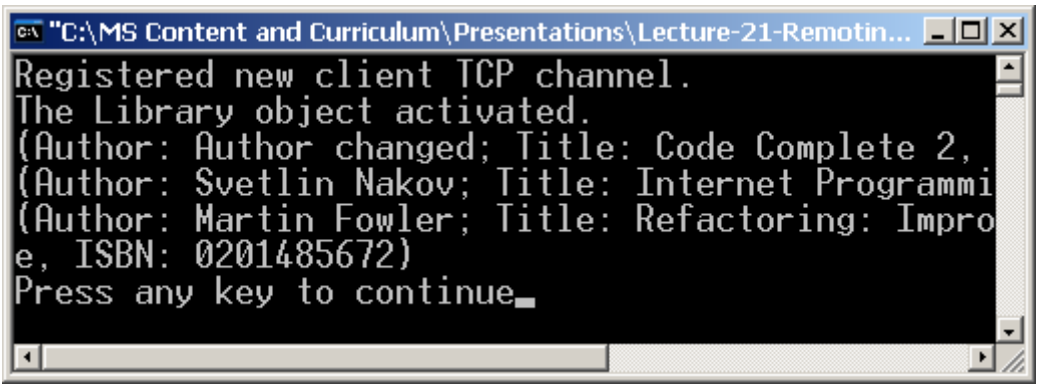
Трябва да обърнем внимание на последните редове на **`Main()`** метода на клиента – тези с цикъла за обхождане на всички книги в библиотеката. Тъй като книгите се маршализират по референция всяко обръщение към техен метод или свойство се реализира като обръщение към сървъра. Когато се извършват по няколко такива в цикъл е възможно да загубим доста от производителността на приложението си. Този цикъл е типично тясно място в изпълнението на програмата и е добре той да бъде оптимизиран. Това може да са осъществи като направим типа **`Book`** да се маршализира по стойност. По този начин клиентът ще работи с локални копия на всяка книга и ще си спестим многото отдалечени извиквания в цикъла. Това решение обаче не трябва да се прилага сляпо навсякъде, защото води до промяна в поведението на обектите от тип **`Book`**, тъй като при промяна на техните свойства се модифицира само локалното им копие при клиента.

Сървърът и клиентът в действие

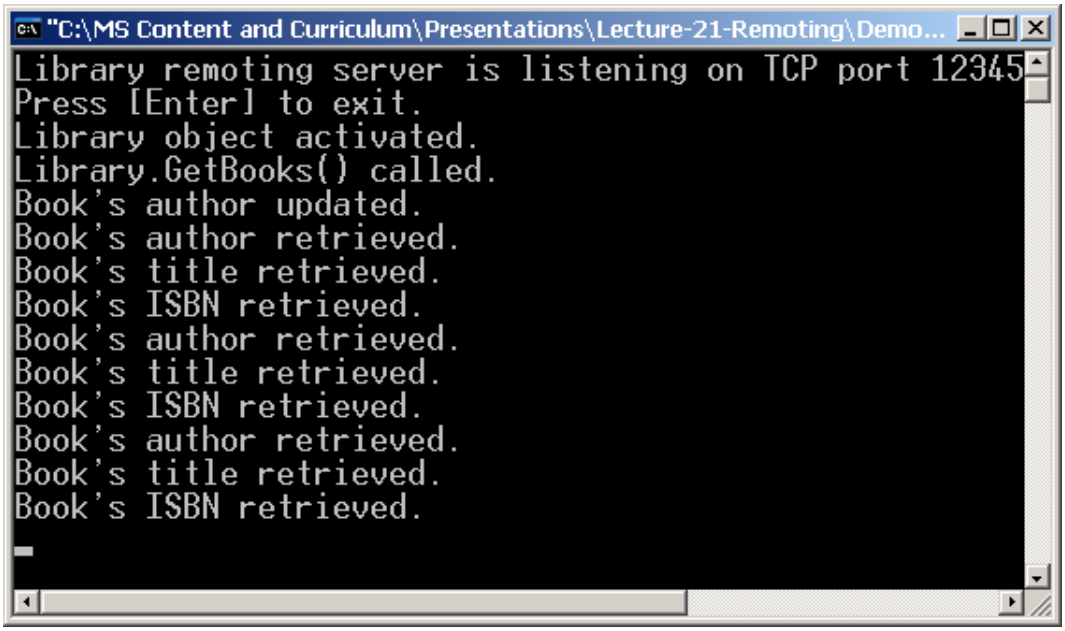
След компилиране и стартиране на сървърното приложение получаваме следния резултат:



Сървърът е успешно стартирал, регистрирал е типа **`Library`** като отдалечен обект и очаква своите клиенти. Ако в този момент компилираме и стартираме клиентското приложение, ще получим следния резултат:



Вижда се, че клиентът успешно е обновил автора на първата книга и е извлякъл от сървъра списъка на всички книги от библиотеката. След приключване на работата на клиента, конзолата на сървъра изглежда по следния начин:



Това показва, че клиентът успешно е активирал **`Library`** обекта, след което е извикал методът му **`GetBooks()`**. След това е обновена една от книгите и информацията за всяка от тях (заглавие, автор и ISBN) е била извлечена. Понеже класът **`Book`** използва маршализация по референция, всеки достъп до свойство от този клас от страна на клиента се изпълнява чрез отдалечено извикване на сървъра.

Пример за маршализация по стойност

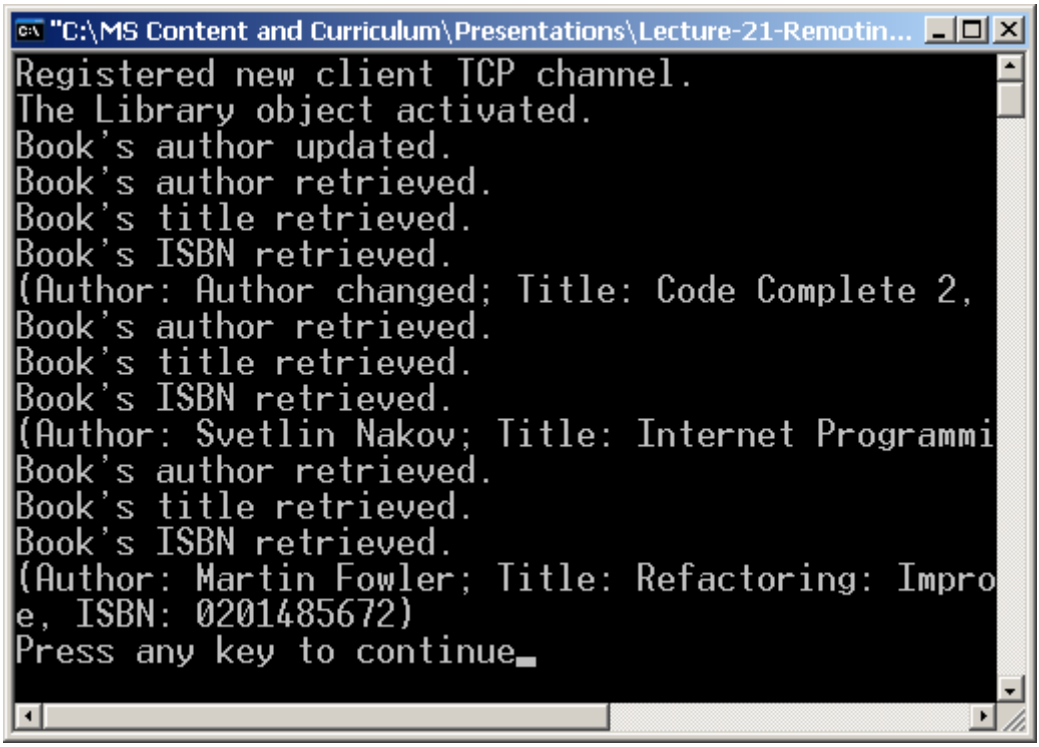
Нека сега направим малка промяна в класа **`Book`**, който се използва от сървъра, и да го направим да се маршализира по стойност. Трябва да заменим реда:

```
public class Book : MarshalByRefObject
```

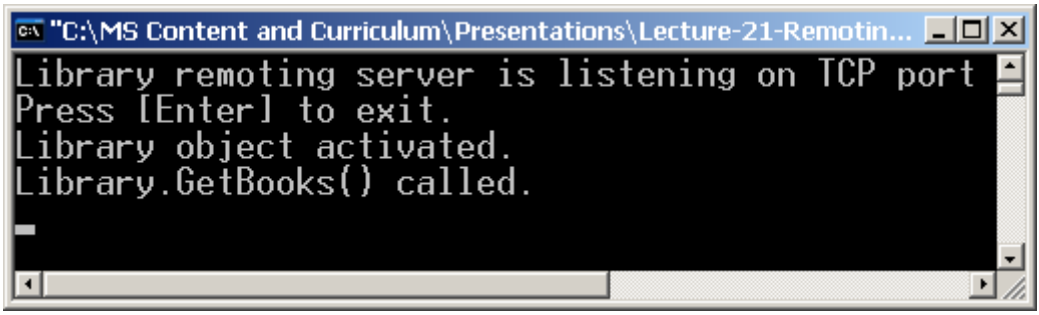
със следния ред:

```
[Serializable] public struct Book
```

Нека прекомпилираме сървъра и пак стартираме първо сървъра, а след това клиента. Конзолата на клиента след успешното му изпълнение изглежда по следния начин:



Ясно се забелязва, че достъпът до свойствата на класа **Book** вече се изпълняват локално при клиента, а не на сървъра. Това е така, защото класът **Book** след промяната се маршализира по стойност и не извършва отдалечено извикване при всеки достъп до негово свойство. Ето как изглежда и сървърът след успешното изпълнение на клиента:



Забелязва се, че въпреки активната работа с обектите от класа **Book**, извлечени чрез метода **GetBooks()**, сървърът не отпечатва нищо при достъпа до техните свойства. Това е така, защото работата с тях се извършва при клиента, понеже тези обекти се маршализират по стойност.

Ако с дебъгера на VS.NET проверим състоянието на книгите на сървъра, ще установим, че авторът на първата книга не е променен, въпреки, че клиентът го променя изрично. Това е така, защото клиентът променя само локалното си копие на този обект (заради маршализацията по стойност).

Проблемът с общите типове

Нещо, което не споменахме изрично за горните примерни сървър и клиент, е че за да се компилират и работят правилно, и клиентът и сървърът трябва да имат копие от асемблито със споделените типове, които се използват. Следващата част е посветена изцяло на този проблем.

Както видяхме в цялата тема, а и от примерите, за да може да работи едно приложение посредством Remoting, трябва и клиентът и сървърът да разполагат с описание на общите за тях типове и техните методи.

Споделено асембли с типове

В .NET Framework типовете се описват от метаданните в асемблитата и затова най-интуитивното решение на проблема с общите типове е да копираме асемблито с типовете данни в директорията на приложението както на сървъра така и на клиента. Този подход има добри и лоши страни.

Добрите са, че всеки разполага с дефинициите на типовете и е възможно е да се организира offline работа с данните.

Лошите страни са, че когато имаме проблем с някой от типовете и направим промени в него (което не е задължително да е предизвикано от проблем!), трябва не само да подменим асемблитата на сървъра, а да накараме всеки един от клиентите да подмени своите асемблита, които са засегнати от промяната. Практиката показва, че това е скъпоструващ, неприятен и сложен процес.

Споделено асембли с интерфейси

Едно частично решение на горния проблем е при клиента да не се разпространяват самите класове (типове), а само интерфейсите, които те имплементират. По този начин имаме възможност да скрием имплементацията на класовете си и всички промени, които не засягат интерфейса на класа, да окажат влияние само върху асемблито с типовете на сървъра. По този начин много по-рядко ще се налага да заставяме клиентите да обновяват своите асемблита, но губим възможността клиентът да може да работи в offline режим. Въпреки това тази практика е препоръчителната и най-често използваната.

Soapsuds.exe

Друг подход за осигуряване на клиента с метаданните, от които се нуждае, е използването на инструмента **soapsuds.exe**. Той се намира в **<директория на VS.NET 2003>\SDK\v1.1\Bin**. Чрез него от готовото асембли с типовете, които ще поставим на сървъра, можем да извлечем само метаданните и да ги компилирате отново в друго асембли, което да използваме при клиента. Този подход не се различава съществено от подхода със споделено асембли, съдържащо общите типове.

Хостинг на Remoting типове в IIS

Един въпрос, свързан с разпространяването на общи типове, който само бегло засегнахме при разглеждането на Remoting сценариите, беше хостингът на асемблита с Remoting типове в IIS.

Remoting инфраструктурата ни позволява да се възползваме от функционалността, която Internet Information Services предлага за хостинг на различни приложения. Такива приложения могат да бъдат уеб приложения, уеб услуги, Remoting приложения и др.

За да използваме IIS за хостинг на Remoting сървърни приложения, трябва да направим 3 неща:

1. Да създадем виртуална директория в IIS.
2. В нея да запишем един Remoting конфигурационен файл, който да има специално име – **Web.config**.
3. Да създадем поддиректория **bin**, в която да копираме асемблитата с типовете, които искаме да използваме като отдалечени.

Ограниченията, които IIS ни налага, са да използваме HTTP протокол. Хоствайки своите отдалечени типове по този начин, ние нямаме нужда да се грижим специално за сигурността и мащабируемостта на сървъра и естествено нямаме нужда да пишем приложение, което да бъде сървър, тъй като за това се грижи IIS.

Remoting приложенията в IIS работят както уеб приложенията и уеб услугите – хостват се и се управляват от сървъра и стартират заедно с него. За тях могат да се настройват сигурността, използваните ресурси и много други неща, които се предоставят от IIS.

Упражнения

1. Обяснете основните концепции на .NET Remoting инфраструктурата – канали, форматери, видове активация, видове маршализация и жизнен цикъл на обектите.
2. Реализирайте клиент-сървър приложение за разговори в реално време (chat), базирано на .NET Remoting. Използвайте TCP канал, бинарен форматер, маршализация по референция и **Singleton** активация от

сървъра. Сървърът трябва да поддържа списък на свързаните към него потребители и да позволява няколко разговора (chat сесии) едновременно. Клиентът (Windows Forms приложение) трябва да може да започва chat сесия, да изпраща съобщения до другите потребители и да затваря chat сесия.

3. Реализирайте клиент-сървър приложение, базирано на .NET Remoting технологията, за обслужване на библиотека с албуми със снимки. Сървърът трябва да поддържа операциите: извличане на албумите, извличане на снимките от всеки албум, добавяне на албум, добавяне на снимка, изтриване на албум, изтриване на снимка, преместване на снимка в друг албум. Албумите не могат да бъдат вложени един в друг. Използвайте файловата система за съхранение на албумите със снимките. Клиентът трябва да е Windows Forms приложение и да предоставя интерфейс към изброените операции. Използвайте HTTP канал, SOAP форматер, активация от клиента и маршализация по стойност. Конфигурирането на клиента и сървъра трябва да става с външен XML файл.

Използвана литература

1. Светлин Наков, .NET Remoting (отдалечено извикване) – http://www.nakov.com/dotnet/lectures/Lecture-21-Remoting-v1.0.ppt
2. MSDN Library, Piet Obermeyer and Jonathan Hawkins, Microsoft .NET Remoting: A Technical Overview – http://msdn.microsoft.com/library/en-us/dndotnet/html/hawkremoting.asp
3. MSDN Library, Paddy Srinivasan, An Introduction to Microsoft .NET Remoting Framework – http://msdn.microsoft.com/library/en-us/dndotnet/html/introremoting.asp
4. MSDN Magazine (12/2003), Juval Lowy, Managing the Lifetime of Remote .NET Objects with Leasing and Sponsorship – http://msdn.microsoft.com/msdnmag/issues/03/12/LeaseManager/default.aspx
5. MSDN Library, Piet Obermeyer and Jonathan Hawkins, Format for .NET Remoting configuration files – http://msdn.microsoft.com/library/en-us/dndotnet/html/remotingconfig.asp

Глава 23. Взаимодействие с неуправляван код

Автор

Мартин Кулов

Необходими знания

- Базови познания за общата система от типове в .NET (Common Type System)
- Базови познания за езика C#
- Базови познания за езика C++
- Базови познания за технологията COM
- Базови познания за програмиране под Win32 със C и C++
- Познания за атрибутите в .NET Framework

Съдържание

- Обща среда или виртуална машина
- Платформено извикване (P/Invoke)
- Преобразуване на данни (marshalling)
- Имплементиране на функция за обратно извикване (callback)
- Взаимодействие с COM (COM interop)
- Извикване на COM обект от управляван код
- Runtime Callable Wrapper (RCW)
- Разкриване на .NET компонент като COM обект
- COM Callable Wrapper (CCW)
- Взаимодействие със C++ чрез IJW

В тази тема ...

В настоящата тема ще разгледаме как да разширим възможностите на .NET Framework чрез употребата на предоставените от Windows приложни програмни интерфейси (API). Ще се спрем на средствата за извикване на функционалност от динамични Win32 библиотеки и на проблемите с преобразуването (marshalling) между Win32 и .NET типовете.

Ще обърнем внимание на връзката между .NET Framework и COM (компонентният модел на Windows). Ще се спрем както на извикването на COM обекти от .NET код, така и на разкриване на .NET компонент като COM

обект. Накрая за любителите на вечния C++ ще разгледаме технологията IJW за използване на неуправляван код от програми, написани на Managed C++.

Какво разбираме под взаимодействие с неуправляван код?

Настоящата книга дава възможност да изучим в детайли действието на .NET Framework и след прочитането й ще знаете да пишете реални приложения. Въпреки сложността и огромния набор от класове, които .NET Framework предоставя, често в ежедневната ни работа се нуждаем от функционалност, която не е вградена в .NET Framework.

Нека си представим следната ситуация – петък следобед е, кротко си обикаляме по любимите сайтове и чакаме колегите да приключат работа, за да може да направим една бърза игра преди края на работния ден. В този момент спокойствието ни се нарушава от ръководителя на проекта, който с леко изнервен тон ни съобщава, че до края на деня трябва да се напише нова функционалност. Приложението трябва да обхожда и да взима състоянието на всички създадени прозорци. Приемаме задачата като се успокояваме, че ще стане за 15 минути понеже сме правили подобно нещо още преди няколко години с помощта на **`EnumWindows(…)`** функцията от Windows API. Няма начин да не е направен такъв еквивалент и в .NET Framework. След кратко ровене из MSDN Library с леко раздразнение откриваме, че такъв метод няма в нито един клас на .NET Framework.

Подобни ситуации са доста често срещани и причината, че не всички методи от Windows API се предоставят от .NET Framework е много проста. От създаването на .NET Framework са минали около 5 години, докато Windows съществува от близо 20 години. Ако трябваше да чакаме .NET Framework да покрие целия набор от функции на Windows API може би едва сега щяхме да използваме първата бета на .NET Framework.

Създателите на .NET Framework са били достатъчно прозорливи, за да разберат, че тайната на успеха на .NET Framework ще се дължи до голяма степен на възможностите за взаимодействие със съществуващия код. Съществува огромно количество неуправляван код, написан до момента. Използването на управляван код има много предимства, но никоя фирма няма да захвърли работещата си програма, само за да я напише отново чрез управляван код. По-скоро фирмата би инвестирала в разработката на бъдещи модули написани чрез управляван код. За да е възможно работата на цялата система, обаче, е необходимо управляваният код да може да "говори" с неуправлявания код. В настоящата тема ще разгледаме какви техники ни предоставя .NET Framework, за да направим възможно взаимодействието на управляван с неуправляван код и обратно.

Обща среда или виртуална машина

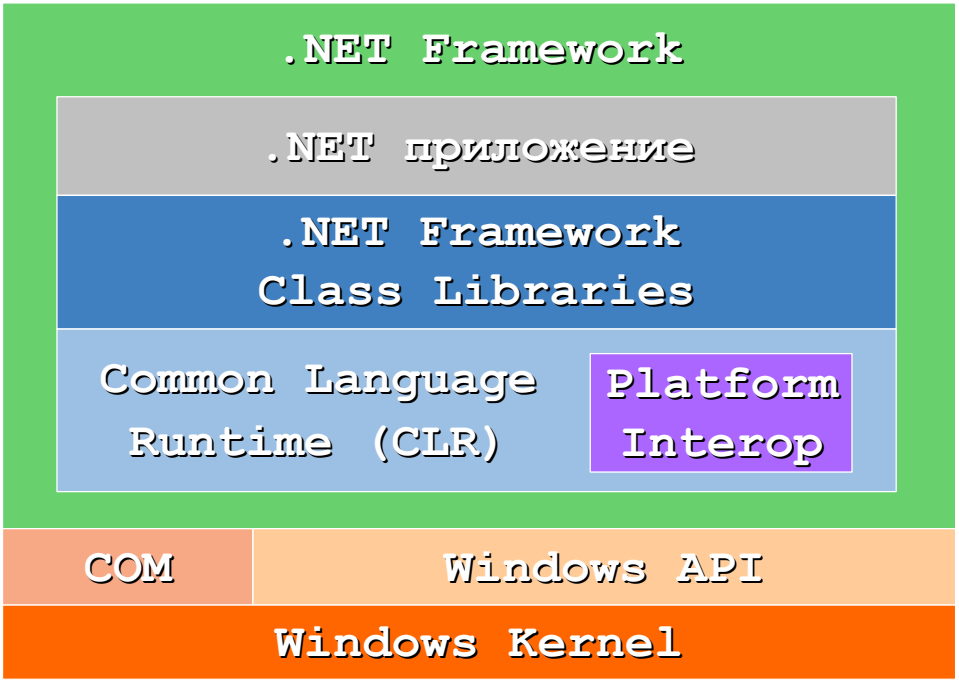
Често като описваме какво представлява .NET, казваме, че е съставен от обща среда за изпълнение. Какво всъщност Майкрософт има предвид под обща среда и как тя се различава от виртуалната машина, която Java

използва? Нека анализираме двата подхода и направим кратка съпоставка между тях.

Среда за контролирано изпълнение .NET CLR (обща среда)

Платформата .NET не цели универсалност спрямо хардуера и операционната система, върху които приложението се изпълнява. Тя е направена за да даде абстракция от операционната система, но операционната система си остава Windows. Кодът написан на CLI (Common Language Infrastructure) съвместим език се компилира до машинно зависим изпълним код за Windows, който го прави толкова бърз колкото е едно C++ приложение написано за Windows. Дори повече, компилацията дава възможност да се оптимизира кодът за съответния процесор, на който ще се изпълнява приложението.

По-долу е представена схематично архитектурата на .NET Framework:



Подобно на CLR, Platform Interop е неразделна част от .NET Framework. Цялата функционалност на Platform Interop се достъпва чрез помощни класове от .NET Framework. Настоящата тема ще ви запознае с тези класове, как те се използват и какви особености са характерни за тях.

Виртуална машина JVM

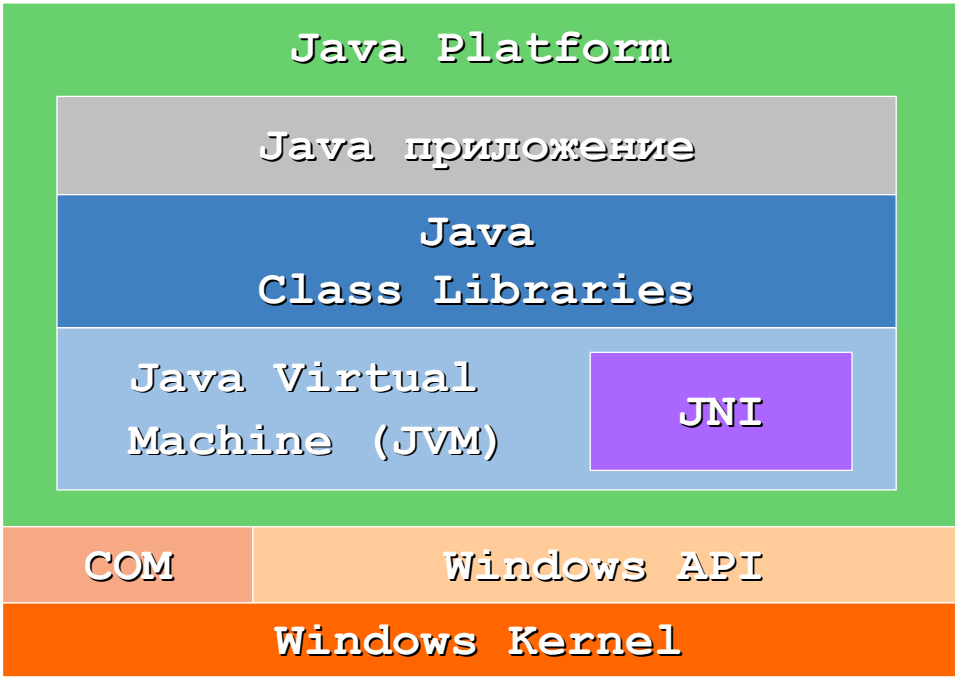
Java платформата, изпълнява програмния си код в специална среда за контролирано изпълнение, наричана Java Virtual Machine (JVM). И при нея

сорс кодът на програмите се компилира до междинен код (т.нар. Java bytecode), който след това се изпълнява от JVM.

Още от създаването на Java платформата основна цел при дизайна на JVM е възможността за изпълнение на програми на езика Java върху различни хардуерни платформи и операционни системи. Това беше амбициозна идея за времето си и в голяма степен Java дължи успеха си на нея.

Разбира се, платформената независимост постави сериозен проблем за програмистите, които искаха да ползват специфични за операционната система услуги напр. DirectX, NTFS, Active Directory и др. За тях е предоставен достъп до части от Windows (или друга ОС) чрез JNI (Java Native Interface). Полученото приложение, което използва JNI технологията, се обвързва с определена операционна система (ОС) и то не може да бъде ползвано на други, освен ако не бъде преправено за новата операционна система.

За сравнение на интеграцията на Platform Interop по-долу е показано как JNI функционалността е интегрирана в JVM:



Аналогично на Platform Interop, достъпването на системни и зависещи от операционната система функции се извършва през класове, които JNI предоставя. За удобство на програмиста JNI също предоставя набор от класове, които позволяват да се предават данни към неуправлявания код като това става прозрачно за извикващия.

Платформено извикване (P/Invoke)

Един от начините за извикване на неуправляван код от .NET Framework е използването на така наречения **P/Invoke**. Наименованието идва от Platform Invoke или в превод – платформено извикване. По същността си този метод изисква да се направят две неща. Да се укаже в кой DLL (Windows библиотека) се намира API функцията, която трябва да се извика и да се опише дефиницията на функцията в управлявания код.

Първата стъпка е необходимо да бъде извършена, за да се определи точният адрес на извикваната функция, но защо е необходимо да се мине през втората стъпка? Защо .NET Framework не генерира автоматично управляван код, който да осигури правилното извикване на API функцията? Причините са две. Първо, за разлика от управлявания код, API функциите не съдържат информация в себе си за броя на параметрите и техния тип. Тази информация се съдържа в библиотеки, които обикновено се разпространяват заедно със средата за разработка. Второ, дори и да предположим, че има начин да се намери броя на параметрите на API функцията и техния тип, определянето на съответстващия тип от CTS за всеки един параметър не е еднозначно. Както ще видим по-долу, съществуват различни възможности за преобразуване на един тип от неуправляван код към тип от Common Type System (CTS).

Описването на API функции в управляван код може да се окаже тежка задача, ако е необходимо да се извика голям набор от API функции. Затова P/Invoke се препоръчва, когато е необходимо да се извика малък брой помощни API функции. Добра практика е дефинициите да се изведат в отделен управляван клас, който би могъл да се преизползва в различни проекти и така да се спести време за писане и отстраняване на проблеми.

Най-добрият ресурс за това как се описва дадена API функция в управляван код може да намерите на адрес www.pinvoke.net [4]. Този сайт съдържа богат набор от по-често използвани функции от Windows API, съответните им .NET дефиниции (на C# и/или VB.NET), както и примерен код, който показва как могат да бъдат извикани през P/Invoke.

Атрибут DllImport

Както видяхме по-горе, за да извикаме избрана от нас API функция е необходимо да укажем в кой DLL се намира тя. Описването се извършва декларативно с помощта на атрибута **`DllImport`**. Този атрибут се прилага само върху методи. С негова помощ се маркира съответния метод от управлявания клас като прокси към неуправлявания код. Методите трябва задължително да бъдат маркирани като **`static extern`**. Указването на желания DLL става при конструирането на атрибута, както е показано в следващия примерен код:

```
[DllImport("user32.dll", EntryPoint="LoadIconW",
   ExactSpelling=true, CharSet=CharSet.Unicode)]
```

```
public static extern IntPtr LoadPredefinedIcon(IntPtr hinst,
   IntPtr icon);
```

По подразбиране, ако не зададем стойност на свойството **EntryPoint**, атрибутът използва името на метода, за да намери API функцията в указания DLL. Атрибутът **DllImport** ни дава възможност да използваме име на API функцията различно от името на метода в управлявания код. Когато изрично указваме името на входната точка в използвания DLL, трябва да се има предвид, че е възможно това име да се различава в Windows 9x и в Windows NT. Причината за това е добавената поддръжка на Unicode в Windows NT и следващите му версии (Windows 2000, XP, 2003, ...).

Когато използвате свойството EntryPoint трябва съвсем точно да сте указали сигнатурата на C# метода. Ако има и съвсем малко несъответствие е възможно P/Invoke да не успее автоматично да намери зададения метод и резултатът от извикването на C# метода ще бъде изключение от типа `System.NullReferenceException` - "Object reference not set to an instance of an object."

ANSI и Unicode версии на API функциите

В зависимост от версията на Windows се използва различен набор от API функции – ANSI и Unicode версия. За да може да се обработят Unicode символни низове е необходимо всеки параметър, указващ символен низ, да бъде деклариран като такъв. В противен случай при опит за четене на ANSI низ от Unicode параметър или обратното най-често ще доведе до прочитане на некоректни данни или дори до грешка от тип ‘page fault’.

В посочения по-горе пример се търси входната точка **LoadIconW** от библиотеката **user32.dll**. Суфиксът ‘**W**’ в случая означава, че трябва да се зареди Unicode версия. Ако се използва суфикс ‘**A**’, напр. **LoadIconA**, .NET Framework ще се обърне към ANSI версията на тази функция. Когато трябва да се напише приложение, което трябва да се компилира както за Unicode, така и за ANSI версия, най-удачно е да се използва свойството **ExactSpelling** като му присвоим стойност **false**. Това свойство определя автоматично версията на API функцията според настройките на проекта и не е необходимо да се указва суфикса на името на функцията. По-надолу ще разгледаме как става преобразуването на символните низове и тяхното кодиране в зависимост от използваната платформа.

Извличане на резултат от API функция

Извличането на резултата от извиканата функция става с помощта на метода **Marshal.GetLastWin32Error**. За целта е необходимо да зададем стойност **true** на свойството **SetLastError**, иначе стойността на върнатия

резултат ще се загуби поради междувременно извикана API функция и то още преди управлението да бъде върнато на управлявания код.

Извличане на резултат от API функция – пример

Нека разгледаме как може да извикаме функцията **FindFirstFile** от библиотеката **kernel.dll**. Тази функция служи за намирането на първия файл отговарящ на дадено търсене. Ще използваме P/Invoke, за да извикаме неуправляваната функция, с помощта на C# метод, който преименува извикваната функция, и ще извлечем резултата от нейното изпълнение.

1. Стартираме VS.NET и създаваме нов проект от тип Class Library с име Interop.
2. Избираме създаденият проект от Solution Explorer и с десен бутон щракаме Add → Add Class и за име на клас избираме **FileManagement**.
3. Към създаденото VS.NET решение добавяме ново конзолно приложение, което ще използваме, за да стартираме класа създаден в предната стъпка.
4. Ето как изглежда кодът на клас **FileManagement** до момента:

```
using System;

namespace Interop
{
   /// <summary>
   /// Summary description for FileManagement.
   /// </summary>
   public class FileManagement
   {
      public FileManagement()
      {
         //
         // Add constructor logic here
         //
      }
   }
}
```

5. Сега добавяме метода **FindFirst()**, който ще направи връзката с неуправлявания код.

```
public static extern IntPtr FindFirst(string wildcard, ref
   FindFirstData fileData);
```

6. Тъй като този метод използва структура, в която връща резултат от търсенето, ще дефинираме една празна структура, която да използваме при декларацията на метода.

```
public struct FindFirstData
{
}
```

7. Добавяме референция към пространството от имена **System.Runtime.InteropServices**.
8. Добавяме атрибута **[DllImport]** към метода от предната точка и инициализираме неговия конструктор с низа "**kernel.dll**". Това е името на файла, в който се намира неуправляваната функция.
9. Задаваме стойност **FindFirstFile** на свойството **EntryPoint** на атрибута **DllImport** за да укажем, какво е името на извикваната неуправлявана функция.
10. Задаваме стойност true на свойството **SetLastError** на атрибута **DllImport**, за да укажем на P/Invoke, че искаме да получим резултатът от изпълнението на функцията.
11. Полученият метод до момента изглежда така:

```
[DllImport("kernel32.dll", EntryPoint="FindFirstFile",
  SetLastError=true)]
public static extern IntPtr FindFirst(string wildcard, ref
  FindFirstData fileData);
```

12. Нека сега да извикаме управлявания C# метод и да отпечатаме резултата от него.
13. Щракаме върху конзолното приложение и добавяме като връзка проектът Class Library, съдържащ класа **FileManagement**.
14. В метода **Main** на конзолното приложение създаваме нова помощна структура **fileData** от типа **Interop.FindFirstData**.
15. Добавяме референция към пространството от имена **System.Runtime.InteropServices**.
16. След като създадем структурата, правим извикване на управлявания метод и записваме резултата от извикването на метода в променлива от тип **IntPtr**.
17. Отпечатваме на екрана резултатът от извикването на **Marshal.GetLastWin32Error**.
18. Ето го резултатния код:

```
[STAThread]
static void Main(string[] args)
{
  Interop.FindFirstData fileData = new Interop.FindFirstData();
  Interop.FileManagement.FindFirst(@"c:\*.*", ref fileData);
```

```
    Console.WriteLine("Error is 0x{0:x}",
    Marshal.GetLastWin32Error());
}
```

19. Щракваме върху конзолното приложение и избираме "Set as StartUp Project".
20. Стартираме решението получаваме резултат 0x7f. Това е кодът за грешка, вследствие от използването на празна структура **FindFirstData**, която направихме само за целите на примера. Като потърсим грешка 0x7f (127 десетично) в MSDN в списъка от резултатите на **GetLastError()** - неуправляван код, ще видим, че тази грешка гласи: "The specified procedure could not be found."
21. За да отстраним грешката, е необходимо да зададем правилна декларация на структурата **FindFirstData**. Такова описание може да намерим на сайта www.pinvoke.net.

Как работи P/Invoke?

Как .NET Framework намира адреса на API функцията, за да предаде управлението на нея? Всеки DLL съдържа списък с имената на дефинираните в него функции. Прочитането на тези имена може да стане с помощта на инструмент, който ще разгледаме след малко.

Когато .NET Framework срещне име на DLL за първи път, зададено в атрибута **DllImport**, този DLL се зарежда в паметта. По името на функцията се извлича точния адрес на кода, който трябва да се изпълни. След като адресът на функцията е намерен, се извършва преобразуване на данните от управляван към неуправляван код и контролът се подава на неуправлявания код. Процесът на преобразуване ще разгледаме малко по-нататък.

Извикването на неуправляван код с P/Invoke може да хвърли изключение, затова проверете изрично как работи извикваната функция при различни състояния на грешка!

Командата DUMPBIN

За да разгледаме всички функции, които даден **.DLL** файл съдържа, може да използваме инструмента **DUMPBIN**. Той се разпространява заедно с MS Visual Studio и се стартира от командния ред. Този инструмент има много възможности, но ние ще се спрем само на една от тях – възможността за преглед на всички външни (exported) функции за посочения DLL. Стартирането на **DUMPBIN** става както е показано в примера:

```
DUMPBIN /EXPORTS C:\WINDOWS\system32\user32.dll
```

```
…
ordinal hint RVA      name
…
 446    1BD 0000CBBB LoadIconA
 447    1BE 000188E3 LoadIconW
…
```

Функциите в даден DLL могат да се достъпват освен по име и по номер. Номерът на всяка функция е показан в колоната **ordinal**, а **name** е името, което се задава на атрибута **[DllImport]**. Полезна информация се съдържа и в колоната RVA (Relative Virtual Address). Това е отместването, на което извикваната функция се намира спрямо началото на зададения DLL. Реалният адрес, на който се намира функцията след като посочения DLL бъде зареден в паметта, е равен на отместването на което е зареден този DLL плюс стойността на RVA.

Възможно е извикването на API функцията да стане по номер, а не по име. За целта на EntryPoint трябва да се присвои номерът от ordinal колоната, предхождан от знака #. Например EntryPoint="#447".

Зареждане на системна икона – пример

Настоящия пример демонстрира как да извикаме неуправлявана функция и да преобразуваме резултата от неуправляван към управляван ресурс. Като резултат ще променим иконата на главния прозорец на Windows Forms приложение.

1. Създаваме ново Windows Forms приложение.
2. Компилираме и стартираме новото приложение.
3. Прозорецът на приложението изглежда подобно на показаното по-долу:

4. От MSDN Library намираме функцията **LoadIcon**, която служи за зареждане на системни икони. Тази функция приема два параметъра. Първият указва адреса на модула, от който зареждаме иконата (в нашия случай се подава NULL), а втория – името на иконата (за системна икона се подава номер резервиран за всяка една икона).
5. Създаваме декларация на метод **LoadPredefinedIcon**, която ще използваме за връзка с неуправляваната функция и й прилагаме атрибута **DllImport**, като по този начин реалното име на неуправляваната функция ще бъде намерено автоматично:

```
[DllImport("user32.dll", EntryPoint="LoadIcon")]
public static extern IntPtr LoadPredefinedIcon(IntPtr hinst,
   IntPtr icon);
```

6. Ако искаме да укажем точното име на неуправляваната функция, трябва да проверим имената на функциите в библиотеката **user32.dll** с помощта на командата **dumpbin**.
7. Отваряме "Visual Studio .NET 2003 Command Prompt" и стартираме командата **dumpbin c:\windows\system32\user32.dll /exports**.
8. В получения списък ще открием две имена, **LoadIconA** и **LoadIconW**, съответно за ANSI и Unicode версията на тази функция.
9. Използваме свойството **ExactSpelling** със стойност **true**, за да укажем, че ще подадем точното име на функцията:

```
[DllImport("user32.dll", EntryPoint="LoadIconW",
   ExactSpelling=true, CharSet=CharSet.Unicode)]
public static extern IntPtr LoadPredefinedIcon(IntPtr hinst,
   IntPtr icon);
```

10. Създаваме константа, която да указва номера на системната икона, която искаме да заредим. Номерата на системните икони се намират във файла **winuser.h** от Platform SDK.

```
static IntPtr IDI_ASTERISK = (IntPtr) 32516;
```

11. След като направихме декларацията за P/Invoke функцията можем да направим реалното извикване.
12. Нека в събитието Load на формата добавим следния код, който извиква зареждането на иконата, преобразува манипулатора на иконата към управляван обект с помощта на статичния метод **Icon.FromHandle()**, и присвоява управлявания обект на свойството **Icon** на главния прозорец.

```
private void MainForm_Load(object sender, System.EventArgs e)
{
   try
   {
      // Get handle of the system icon
      IntPtr hicon =
         LoadPredefinedIcon(IntPtr.Zero, IDI_ASTERISK);

      if (hicon != IntPtr.Zero)
      {
         // Create new object from the retrieved handle
         Icon icon = Icon.FromHandle(hicon);
```

```
            // Change the icon of the main window
            this.Icon = icon;
        }
    }
    catch(Exception exc)
    {
        Debug.WriteLine("Exception: " + exc.Message);
    }
}
```

13. Освобождаване на системни икони не се прави. Поради тази причина в примера няма код, който да извиква **`Dispose()`** метода на променливата **`icon`**.

14. Когато зареждаме икона от даден файл, трябва да имаме предвид, че след като свършим работата със заредената икона, трябва да извикаме метода **`Dispose()`** или неуправляваната функция **`DestroyIcon`**, за да освободим системните ресурси.

15. След като стартираме приложението ще видим, че сме променили успешно иконата на главния прозорец:

Преобразуване на данни (marshalling)

Както стана ясно по-горе, при преминаване от управляван към неуправляван код се налага преобразуване на типовете. Това се прави понеже неуправляваната среда не знае нищо за това как да обработи типовете на управляваната среда и обратно. Например в едно C++ приложение, което използва неуправляван код, се налага да използваме Array обект върнат от управляван код. Очевидно е, че този Array обект трябва да се запише в неуправляван масив. Какъв да е типът на този масив обаче? Дали да е масив от цели числа или масив от дробни числа? А дали не съдържа символни низове? А може би съдържа обекти от даден клас, за който неуправлявания код няма описание? Как тогава ще получим и обработим резултата?

За основните типове от CTS се прави автоматично преобразуване към неуправляван тип и обратно. Част от тях са показани в табл. 1. Разбира се, автоматичното преобразуване може да се променя, когато преобразуваните данни са по-сложни. Това би се наложило и в случай, когато преобразуването по подразбиране заема много памет.

За да разберете разликите при предаването на структури и класове е необходимо да сте наясно с термините опаковане и разопаковане (boxing и unboxing), които са описани в темата за Common Type System (CTS).

В настоящата тема ще разгледаме правилата за преобразуване на типовете между двете среди и ще разгледаме няколко примера:

Неуправляван тип	Управляван тип
HANDLE	System.IntPtr
BYTE	System.Byte
WORD	System.UInt16
DWORD	System.UInt32
FLOAT	System.Single
LPSTR, LPCSTR, LPWSTR, LPCWSTR	System.String или System.StringBuilder

Преобразуване на структури

За структурите е известно, че са стойностни типове, както е описано в темата в темата за Common Type System (CTS). Когато една структура бъде опакована, достъпът до нея се осъществява посредством указател, тъй като структурата се намира в динамичната памет.

Аналогично при неуправляван код структурите също се съхраняват в стека и са стойностни типове. Адресът на тази структура може да се получи чрез оператор за извличане на адрес (reference operator), но за разлика от структурата в управлявания код, тя не се опакова и съответно не се мести от стека в динамичната памет.

Съхранението и достъпът до структурите, макар и привидно да са еднакви, са реализирани по различен начин в управляван и неуправляван код. Въпреки това е удобно да се разглежда, че при преобразуване на стойностен тип от едната среда се получава стойностен тип в другата, а при преобразуване на указател към структура се получава опакована структура и се предава с помощта на **ref** параметър.

Таблицата по-долу показва как става преобразуването на структура от неуправляван код към управляван код и обратно.

```
DLLFunc(POINT x) ↔ ManagedFunc(POINT x)
DLLFunc(POINT* x) ↔ ManagedFunc(ref POINT x)
```

Функцията **DLLFunc()** представлява примерна функция от неуправляван код, а **ManagedFunc()** е съответната декларация на C#.

Разполагане на полетата от структурата

Възможно е, когато преобразуваме структура от неуправляван към управляван код, да се установи, че някои полета липсват в едната от структурите, а други не трябва да се преобразуват, защото са твърде големи и ще натоварят излишно приложението. Общата среда за изпълнение (CLR)

дава възможност да се укаже съответствието между полетата на изходната и крайната структура. Това става чрез използването на атрибута **StructLayout**.

Атрибутът StructLayout

Прилагането на **StructLayout** става върху само върху структури и класове. При конструирането на атрибута се задава как да се разположат полетата в структурата в паметта. Когато е необходимо да се укаже точното отместване на всяко поле от началото на структурата, се използва стойността **LayoutKind.Explicit**. Когато искаме да запазим реда на отместването на различните полета, може да се използва стойността **LayoutKind.Sequential** и средата ще подреди полетата в паметта по реда на деклариране им. Това е стойността по подразбиране, която се прилага върху структурите, тъй като е по-вероятно те да се използват при P/Invoke извиквания. Има и трета стойност при конструирането на атрибута – **LayoutKind.Auto**. Това е стойността по подразбиране за класовете. Когато използвате тази стойност, .NET Framework автоматично избира как да подреди полетата от структурата с оглед на намаляването на големината на класа и оптимизиране на работата на системата за управление на паметта. Тази стойност обаче, не позволява преобразуването на структура към неуправляван код.

Атрибутът StructLayout – пример

В следващия примерен код е илюстрирано как се използва атрибута **StructLayout**:

```
[StructLayout(LayoutKind.Explicit)]
public struct SYSTEM_INFO
{
   [FieldOffset(0)]
   public UInt16 ProcessorArchitecture;

   [FieldOffset(4)]
   public UInt32 PageSize;

   [FieldOffset(16)]
   public UInt32 ActiveProcessorMask;

   [FieldOffset(20)]
   public UInt32 NumberOfProcessors;
}

[DllImport("kernel32.dll", EntryPoint="GetNativeSystemInfo")]
private static extern void _GetNativeSystemInfo(
   ref SYSTEM_INFO sysInfo);
```

В примера се използва изрично посочване на отместването на полетата в структурата. Конкретната стойност на отместването на полето се подава

чрез атрибута **FieldOffset**. Този атрибут се прилага само върху полета и няма други свойства. Декларираната структура **SYSTEM_INFO** се използва за извикване на Windows API функцията **GetNativeSystemInfo**, която е изведена чрез атрибута **DllImport** от библиотеката **kernel32.dll**.

Преобразуване на класове

Принципът за преобразуване на класове е аналогичен на този при структурите. За разлика от тях обаче, класовете не се предават по стойност, а по адрес. Тази особеност позволява да се извърши преобразуване на двоен указател от неуправляван към управляван код и обратно. В по-горната таблица беше показано как се предава структура чрез **ref** параметър. Как обаче ще се извърши преобразуването, ако за параметър в неуправлявания код се подаде адреса на адреса на преобразуваната структура?

В неуправляван код използването на адреса на адреса на дадена структура се налага, когато извикващия не заделя памет за тази структура, а това се прави от извиквания метод. Извикващият подава мястото, където да се запише адреса на заделената от извиквания памет на структурата.

На пръв поглед би било логично да напишем следното:

```
DLLFunc(POINT** x)    ↔    ManagedFunc(ref ref POINT x) – ГРЕШНО
```

Такава конструкция, обаче, не е позволена в C#. Най-удачно в този случай е да се използват класове. Те се създават в динамичната памет и дават още едно ниво на адресиране. Чрез тях става възможно преобразуването на параметри, които описват двойни указатели. В този смисъл, когато използваме класове вместо структури, таблицата по-горе се представя по следния начин:

```
DLLFunc(Job* x)     ↔    ManagedFunc(Job x)
DLLFunc(Job** x)    ↔    ManagedFunc(ref Job x)
```

Класовете, които се използват при преобразуването също имат член променливи, за които е нужно да се укаже реда на преобразуването им. Затова атрибута **StructLayout** може да се прилага както върху класове, така и върху структури.

Преобразуване на низове

Преобразуването на примитивните типове в .NET към неуправляван код е сравнително лесно поради наличието на еквивалентни типове в управляван и неуправляван код. При символните низове обаче преобразуването има няколко особености, които ще разгледаме сега.

Низовете са неизменяеми

Първа съществена особеност е, че низовете в .NET са неизменяеми (immutable). Накратко това означава, че веднъж създаден един **String** обект не може да бъде променян. Повече информация за това може да получите от темата за символни низове.

Низовете наследяват System.Object

Друга особеност е, че символният низ се представя чрез обект наследен от **System.Object**. Като се вземе предвид, че един символен знак може да е представен в паметта с един или няколко байта, в зависимост от кодировката на символния низ, преобразуването на един символен низ става трудоемка задача. Въпреки това .NET Framework предоставя такова преобразуване по подразбиране и дава възможност да се настройва как да се извърши преобразуването. Допълнителната настройка става чрез използването на атрибута **MarshalAs** и се налага най-често в случаите, когато кодировките на низа в двете среди (управлявана и неуправлявана) са различни или е необходимо да се укаже големината на преобразувания низ.

String или StringBuilder?

При получаване на символен низ като резултат от извикването на неуправляван код първото нещо, което може би ще направите е да дефинирате поле от тип **String**, което да получи резултата. Трябва да запомните, че поради неизменяемостта на низовете, всяка промяна на текущата стойност на низа е неправилна и може да доведе до грешка с работата със **String** обекта (символен низ в управляван код се променя като се копира неговото съдържание в нов низ). Затова когато е необходимо да се върне символен низ от неуправляван код се използва класът **StringBuilder**. Средата извършва автоматичното преобразуване на неуправлявания символен низ към обекта **StringBuilder**.

Преобразуване на низове чрез StringBuilder

Примерът по-долу показва как се използва **StringBuilder** в една малка програма:

```
using System;
using System.Text;
using System.Runtime.InteropServices;

public class GetComputerNameExample
{
   [DllImport("kernel32")]
   static extern bool GetComputerName(StringBuilder name,
     ref int len);

   static void Main(string[] args)
```

```
    {
        StringBuilder computerName = new StringBuilder(255);

        int len = computerName.Capacity - 1;
        GetComputerName(computerName, ref len);

        Console.WriteLine(computerName);
    }
}
```

В примера се създава обект от тип **StringBuilder** инициализиран с първоначално място за 255 символа. Методът **GetComputerName()** получава този **StringBuilder** обект и неговата големина, и записва връщания резултат в него. Резултатът получен в **StringBuilder** обекта се отпечатва на екрана.

Предаването на символен низ към неуправляван код става по-интуитивно като се използва класа **String**, върху който се прилага атрибута **MarshalAs** както е описано в следващата част.

Атрибут MarshalAs

Атрибутът **MarshalAs** играе основна роля при указването на желания формат на преобразуването. Прилага се върху параметри, член-променливи или резултат от даден метод. Неговата употреба не е задължителна. Ако не се укаже атрибута **MarshalAs**, се използва преобразуването по подразбиране за съответния тип. При конструирането на атрибута се указва към какъв тип неуправляван низ ще се извършва преобразуването. Ако този низ се предава по стойност, т.е. пази се в стека на неуправлявания код, е необходимо да се зададе и големината на низа. За тази цел се използва константата **SizeConst**. По-долу е даден пример как се използва атрибута **MarshalAs**:

Неуправляван код	Управляван код

`struct STOCK`
`{`
	`[StructLayout(LayoutKind.Sequential,`
	`  CharSet=CharSet.Auto)]`
	`struct STOCK`
	`{`

`  TCHAR  ID[32];`	`  [MarshalAs(UnmanagedType.ByValTStr,`
	`    SizeConst=32)]`
	`  public String ID;`

`  Char*  Name;`	`  [MarshalAs(UnmanagedType.LPStr)]` `  public String Name;`

```
  WCHAR* Location;
}
```

```
  [MarshalAs(UnmanagedType.LPWStr)]
  public String Location;
}
```

В примера се показва как структурата **STOCK**, която съдържа различни по вид и кодиране символни низове се описва с помощта на **MarshalAs** атрибута. Обърнете внимание как се използва **SizeConst** полето при преобразуване на неуправляван символен низ с предварително известен размер.

Друг важен елемент от даденият пример е задаването на полето **CharSet** на атрибута **StructLayout**. В примерът е използвана стойност **CharSet.Auto**, която осигурява използването на необходимата кодировка в зависимост от операционната система. Употребата на тази стойност ще окаже влияние единствено върху полето **ID** тъй като за него е посочено, че ще преобразува тип **UnmanagedType.ByValTStr**, който подобно на неуправляваният тип **TCHAR**, е платформено зависим (Unicode – за Windows NT, Windows 2000, Windows XP, и Windows Server 2003; ANSI – за Windows 98 и Windows Me). Ако променим стойността на **CharSet** с **CharSet.Ansi** или **CharSet.Unicode** ще изменим кодирането на полето **ID** съответно към ANSI или Unicode независимо от използваната платформа.

Имплементиране на функция за обратно извикване (callback)

Функцията за обратно извикване служи, както говори името й, за осъществяване на обратна връзка между извикващата и извикваната функция. За най-прост пример може да посочим API функцията **SetWaitableTimer()**, която приема за параметър към коя функция да се обърне след като изтече зададения от извикващия интервал. Функциите за обратно извикване не са нищо ново в света на Win32 програмирането, затова Майкрософт са предоставили лесен начин за тяхната поддръжка в .NET Framework. Подобно на Win32 кода, в който трябва да дефинираме метод отговарящ да определена декларация, така и в .NET Framework е необходимо да имплементираме метод, който отговаря на декларацията определена с помощта на **delegate**. Така, ако вземем за пример функцията за обратна връзка **EnumWindowsProc**, която се използва от Windows API функцията **EnumWindows** и има следната декларация:

```
BOOL CALLBACK EnumWindowsProc(HWND hwnd, LPARAM lParam);
```

В управлявания код е необходимо да декларираме делегат със съответното описание на параметрите. Например:

```
public delegate bool CallBack(int hWnd, int lParam);
```

По този начин при конструирането на такъв делегат се задава метода от управлявания код, който да се изпълни. За повече информация вижте темата за делегати и събития.

Преобразуване на данни – пример

Сега ще разгледаме пример, който демонстрира различни видове на преобразуване на данни.

1. Отваряме решението **Demo-2-Marshalling.sln**.
2. Отваряме файла **MarshallingDemo.cs**.
3. Нека разгледаме съдържанието на **Main** метода:

```
static void Main(string[] args)
{
   // Get and print full path to current executable
   string moduleFullPath = Module.GetFullPath();
   Console.WriteLine("Executable path: {0}\n", moduleFullPath);

   // Get and print hardware configuration
   Machine.SYSTEM_INFO sysinfo = Machine.GetSystemInfo();
   Console.WriteLine("Processor architecture: {0}",
      sysinfo.ProcessorArchitecture);
   Console.WriteLine("Page size and granularity of page " +
     "protection: {0}", sysinfo.PageSize);
   Console.WriteLine("Processors' mask: {0}",
      sysinfo.ActiveProcessorMask);
   Console.WriteLine("Number of processors: {0}",
      sysinfo.NumberOfProcessors);
   Console.WriteLine("Processor type: {0}",
      sysinfo.ProcessorType);
   Console.WriteLine("Virtual memory granularity: {0}",
      sysinfo.AllocationGranularity);

   // Get and print titles of all windows
   string[] desktopWindowsTitles =
      Window.GetDesktopWindowsCaptions();
   Console.WriteLine("\nDesktop windows titles:");
   foreach(string title in desktopWindowsTitles)
   {
      Console.WriteLine(title);
   }
}
```

4. В първата част на метода извличаме и отпечатваме пълния път на текущия изпълним файл. Това става като извикваме методът **GetFullPath** на класа **Module** от файла **Module.cs**.
5. Отваряме файла **Module.cs**.

6. Декларираме метода **_GetModuleFileName()**:

```
[DllImport("kernel32.dll", EntryPoint="GetModuleFileName",
  ExactSpelling=false, CharSet=CharSet.Auto)]
private static extern UInt32 _GetModuleFileName(IntPtr hModule,
  StringBuilder lpFileName, UInt32 nSize);
```

7. Нека разгледаме метода **GetFullPath()**:

```
public static string GetFullPath()
{
  StringBuilder modulePath = new StringBuilder(MAXPATH);
  UInt32 uiSize = _GetModuleFileName(IntPtr.Zero, modulePath,
    (uint) modulePath.Capacity + 1);
  modulePath.Length = (int) uiSize;

  return modulePath.ToString();
}
```

8. Първо създаваме нова инстанция на класа **StringBuilder** с капацитет **MAXPATH**.

9. След извикването на метода **_GetModuleFileName()** като резултат получаваме в **uiSize** колко е голям низа върнат от неуправлявания код.

10. Тъй като за управлявания код не е известно колко голям е върнатия низ, трябва да зададем на свойството **Length** получената големина от **uiSize**.

11. Във втората част от **Main** извикваме **Machine.GetSystemInfo()** за да получим информация за процесора и наличната памет.

12. Отваряме файла **Machine.cs**.

13. В този файл по аналогичен начин както в **Module.cs** извикваме неуправляваната функция **GetNativeSystemInfo()**:

```
public static SYSTEM_INFO GetSystemInfo()
{
  SYSTEM_INFO sysInfo = new SYSTEM_INFO();
  _GetNativeSystemInfo(ref sysInfo);

  return sysInfo;
}
```

14. Функцията **GetNativeSystemInfo** приема като параметър адреса на структурата, в която ще запише резултата, затова трябва да подадем структурата по адрес (с модификатор **ref**):

```
[DllImport("kernel32.dll", EntryPoint="GetNativeSystemInfo")]
private static extern void _GetNativeSystemInfo(
   ref SYSTEM_INFO sysInfo);
```

15. Полученият резултат записваме в структурата **SYSTEM_INFO**, за която сме указали, че се разполага последователно в паметта с атрибута **[StructLayout(LayoutKind.Explicit)]**.

16. Първите две полета от тази структура представляват union по дефиницията на неуправляваната структура, затова те се разполагат на едно и също отместване в структурата с атрибута **[FieldOffset(0)]**:

```
[StructLayout(LayoutKind.Explicit)]
public struct SYSTEM_INFO
{
   [FieldOffset(0)] public UInt32 OemId;
   [FieldOffset(0)] public UInt16 ProcessorArchitecture;
   [FieldOffset(4)] public UInt32 PageSize;
   [FieldOffset(16)] public UInt32 ActiveProcessorMask;
   [FieldOffset(20)] public UInt32 NumberOfProcessors;
   [FieldOffset(24)] public UInt32 ProcessorType;
   [FieldOffset(28)] public UInt32 AllocationGranularity;
}
```

17. В третата част на метода **Main()** извикваме метода **Window.GetDesktopWindowsCaptions()**, който ни връща имената на всички прозорци създадени на текущия работен плот (desktop) като масив от низове.

18. Отваряме файла **Window.cs**.

19. В него имаме дефинирани два P/Invoke метода:

```
[DllImport("user32.dll", EntryPoint="EnumDesktopWindows",
   ExactSpelling=false, CharSet=CharSet.Auto, SetLastError=true)]
private static extern bool _EnumDesktopWindows(IntPtr hDesktop,
   EnumDelegate lpEnumCallbackFunction, IntPtr lParam);

[DllImport("user32.dll", EntryPoint="GetWindowText",
   ExactSpelling=false, CharSet=CharSet.Auto, SetLastError=true)]
private static extern int _GetWindowText(IntPtr hWnd,
   StringBuilder lpWindowText, int nMaxCount);
```

20. Методът **_EnumDesktopWindows()** обхожда всички създадени прозорци по подаден работен плот и извиква callback функция за обработка на резултата.

21. Методът **_GetWindowText()** връща заглавието на прозорец по зададен манипулатор.

22. За дефиниране на callback функция използваме делегата **EnumDelegate**:

```
private delegate bool EnumDelegate(IntPtr hWnd, int lParam);
```

23. Нека разгледаме същинското извикване на метода **_EnumDesktopWindows()**:

```
/// <summary>
/// Get titles of all main windows
/// </summary>
public static string[] GetDesktopWindowsCaptions()
{
   mTitlesList = new ArrayList();
   EnumDelegate enumfunc = new EnumDelegate(EnumWindowsProc);
   IntPtr hDesktop = IntPtr.Zero; // current desktop
   bool success = _EnumDesktopWindows(hDesktop,
      enumfunc, IntPtr.Zero);

   if (success)
   {
      // copy result to array of strings
      string[] titles = new string[mTitlesList.Count];
      mTitlesList.CopyTo(titles);
      return titles;
   }
   else
   {
      // get last error code
      int errorCode = Marshal.GetLastWin32Error();

      string errorMessage = String.Format(
         "EnumDesktopWindows failed with code {0}.", errorCode);
      throw new Exception(errorMessage);
   }
}
```

24. Създаваме инстанция на делегата **EnumDelegate** от метода **EnumWindowsProc()**.

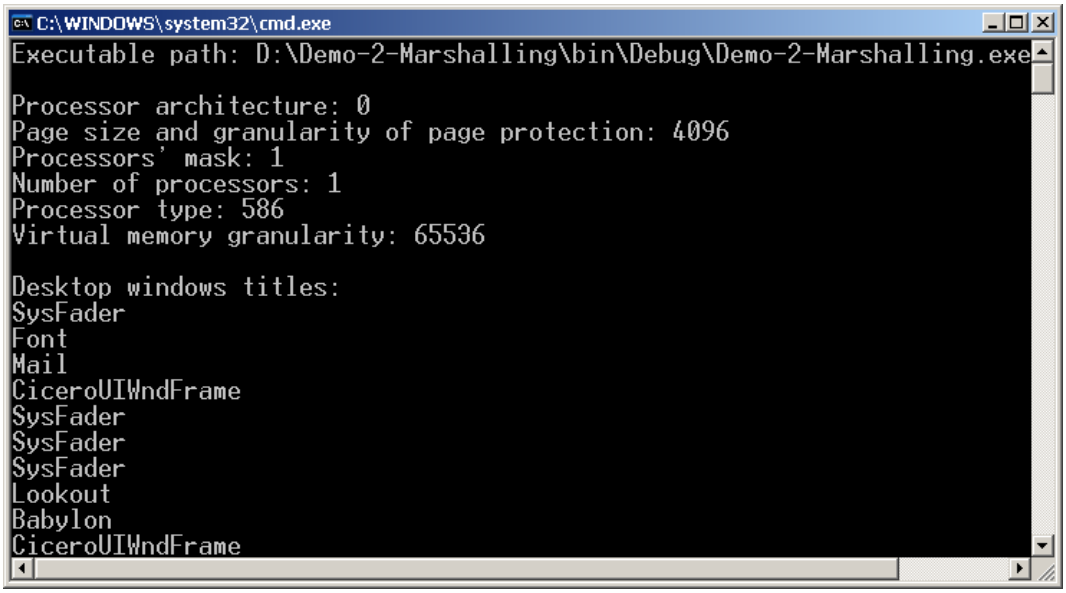
25. Методът **EnumWindowsProc** взима заглавието на подадения прозорец като използва помощния метод **GetWindowText()** и го добавя към общ списък **mTitlesList**:

```
private static bool EnumWindowsProc(IntPtr hWnd, int lParam)
{
   string title = GetWindowText(hWnd);
   if (title.Length > 0)
   {
      mTitlesList.Add(title);
```

```
   }

   return true;
}

/// <summary>
/// Get window title from HWND.
/// </summary>
public static string GetWindowText(IntPtr hWnd)
{
   StringBuilder title = new StringBuilder(MAXTITLE);
   int titleLength = _GetWindowText(hWnd, title,
      title.Capacity + 1);
   title.Length = titleLength;

   return title.ToString();
}
```

26. Извикваме метода **_EnumDesktopWindows()** като за първи параметър му подаваме стойност **IntPtr.Zero**, което означава, че искаме да обходим прозорците на текущия работен плот. За втори параметър подаваме инстанцията на делегата **EnumDelegate** – **enumfunc**.
27. При неуспех използваме метода **Marshal.GetLastWin32Error** за да извлечем системната грешка.
28. При успех копираме съдържанието на списъка в масив от низове и го връщаме на метода **Main**.

Ето как изглеждат първите няколко реда от изпълнения пример:



Взаимодействие с COM (COM interop)

След като се запознахме по какъв начин можем да извикваме функции от операционната система посредством P/Invoke и как да маршализираме прехвърляните данни, нека сега разгледаме някои по-сложни начини за взаимодействието с неуправляван код, а именно работа с COM обекти.

Какво е COM?

Основна задача на .NET Framework е да осигури преизползваемост на създадените компоненти от една система в друга, независимо от използвания език за програмиране. Преди появата на .NET Framework Майкрософт опитаха да решат проблема с преизползваемостта с помощта на COM (Component Object Model). COM е обектен модел за създаване и извикване на компоненти. Той се използва и до днес, но има своите предимства и недостатъци. COM позволява отделянето на отделни модули в компоненти, които могат да се използват от други приложения, независимо от езика, с който са създадени. Всеки компонент заявява своята функционалност, чрез дефинирането на интерфейс. Извикващият код може да разпознае този интерфейс още на ниво компилация или да използва динамично разпознаване интерфейса по време на изпълнение на програмата.

Като всичко хубаво, обаче COM има някои недостатъци. Част от тях са използването на неуправляван код т.е. липса на механизъм за събиране на боклука, възможност за препълване на буфери и др. COM има и проблеми с производителността при използването в големи многонишкови системи, особено когато се осъществява извикване между компоненти създадени в различни апартаменти.

Още при създаването на един COM обект се указва какъв модел за достъп позволява обекта. Този модел за достъп се нарича още "апартамент". Съществуват няколко вида апартаменти – STA (single threaded apartment), MTA (multithreaded apartment) и TNA (thread neutral apartment). Извикването на метод на COM обект, който е създаден в апартамент различен от този на извикващия, с някои изключения, е времеотнемаща задача поради нуждата от извикване на допълнителен синхронизиращ код.

Видове COM обекти и регистрация

Според начина на изпълнение COM обектите са два вида – in process и out of process. Когато COM обектът се изпълнява в адресното пространство на извикващия процес, този COM обект е от тип in process. Най-често такъв COM обект се запазва в DLL файл. За да може да бъде открит този DLL файл като файл съдържащ COM обект е необходимо той да бъде регистриран в Windows Registry. Регистрацията става по съвсем лесен начин като

се използва инструмента **regsvr32 /i <име на файла>**. Премахването на COM обекта от Registry става с параметъра **/u**.

Възможно е да се създаде COM обект, който да се изпълнява в собствено адресно пространство т.е. в отделен процес. Такива COM обекти се наричат 'out of process' и се съхраняват в EXE файлове. За тях също е необходима регистрация, но тя се извършва без допълнителен инструмент, а чрез стартиране на EXE файла с параметър **/regserver**. Премахването на регистрацията става съответно с параметъра **/unregserver**.

Структура на COM обектите

Работата на COM обектите се базира на договори (контракти). Всеки COM обект обявява своята функционалност посредством интерфейс. Веднъж след като обяви даден интерфейс COM обекта гарантира, че няма да променя този интерфейс. Ако се наложи да се добави или промени дефиницията на някой метод, COM обекта трябва да предостави нов интерфейс. При промяна на вече съществуващ интерфейс би настъпила разлика между очакваната и реално използваната дефиниция на интерфейса и приложението, което използва този интерфейс може да спре да работи. Това правило за запазване на дефиницията на вече публикуван интерфейс осигурява контракт, чрез който се постига една основна цел при COM архитектурата – независимост при обновяване на версията.

Интерфейсът IUnknown

По своята същност COM обектите могат да бъдат достъпвани от всеки един процес или нишка. Когато бъде извикан, COM обектът, най-общо казано, предоставя указател към използваната от него памет. До момента в който извикващия приключи своята работа с COM обекта, тази памет не трябва да бъде освобождавана. Тъй като единствено извикващия знае кога е приключил работата с COM обекта, той трябва по някакъв начин да съобщи за това. Поради тази причина всеки COM обект имплементира задължително интерфейса **IUnknown**. Чрез него се следи броя на връзките на създадените връзки към обектите (т.нар. reference counting). Интерфейсът **IUnknown** дефинира два метода, с които се извършва отброяването на връзките. Това са **AddRef** и **Release** методите, съответно за отбелязване и за премахване на връзка. При извикването на **AddRef** и **Release** COM обектът е длъжен да осигури поддръжката на вътрешен брояч за броя на връзките. При достигане на нула, когато бъде извикан метода **Release**, COM обекта може да освободи заеманата от него памет тъй като към него няма активни връзки и извикването му не е възможно.

Интерфейсът **IUnknown** дефинира още един важен метод – **QueryInterface**. Той служи за проверка дали COM обекта поддържа даден интерфейс. При наличие на такъв интерфейс **QueryInterface** връща указател към него, в противен случай резултатът е **NULL**.

Интерфейсът IDispatch

Подобно на C#, COM обектите позволяват силно типизиран достъп до методите, които предоставят. Това улеснява работата с тях и намалява възможността за възникване на грешки от косвеното преобразуване на един тип към друг, което би възникнало ако се ползва слабо типизиран достъп.

За всички, които малко или много са работили с компютър, е известно, че съществуват едни езици като Visual Basic, VBScript, JScript, наричани скриптови (scripting или Automation) езици. Тези Automation езици работят със слабо типизиран достъп, което дава възможност по време на изпълнение на програмата да се определят методите на извикваните обекти и техните параметри. За разлика от силно типизираните езици, голяма част от Automation езиците не се компилират и поради тази причина те получават информация за извиквания обект едва в момента на извикването му.

За да се преодолеят различията между COM и Automation езиците се въвежда един специален интерфейс **`IDispatch`**. Той позволява динамично да бъдат открити методите и съответните параметри на COM обекта, който имплементира този интерфейс. Скриптовите езици проверяват за наличието на този интерфейс, преди да се обърнат към дадения COM обект, проверяват дали търсения метод съществува и дали е възможно да се преобразуват подадените параметри към параметрите на извиквания метод.

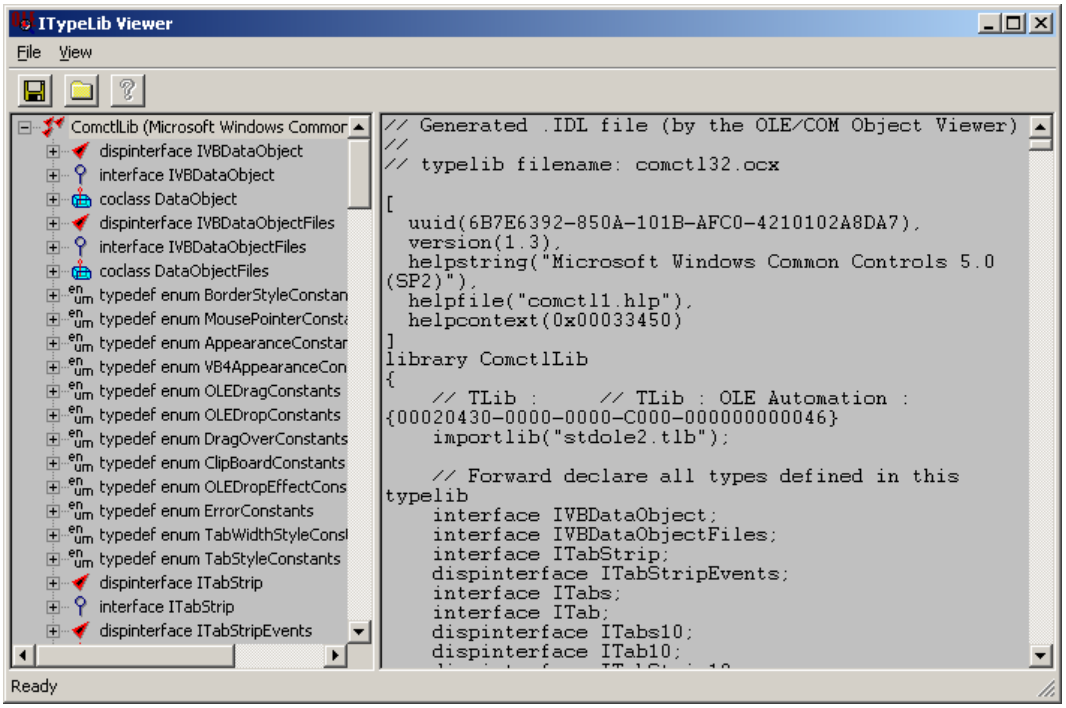
Основните методи, които **`IDispatch`** предоставя са **`GetIDsOfNames()`** и **`Invoke()`**. Подобно на техниката "отражение" (reflection) в .NET Framework методът **`Invoke()`** позволява динамично, по време на изпълнение на програмата, да се извика метод и да се извлече получения резултат.

Извикване на COM обект от управляван код

Както споменахме по-горе, една от главните задачи на Майкрософт при създаването на управляваната среда е била да осигури лесен и интуитивен начин за достъп до неуправляван код. За целта Майкрософт въвеждат използването на Interop асемблии. Това е асемблии, което служи като мост между неуправляваната и управляваната среда. С негова помощ се осъществява преобразуването на данни и дефинирането на помощни типове и класове за улеснение на "средностатистическата кирка" (б.а. кирка = копач, програмист висящ постоянно пред компютъра и извършващ монотонно един и същи тип задачи).

Подобно на .NET асемблитата, COM обектите също имат собствено описание на типовете, методите и класовете, които съдържат. В COM обектите това описание се записва в така наречената типова библиотека (type library). Типовата библиотека може най-лесно да се разгледа с инструмента OLE-COM Object Viewer, който се разпространява с Microsoft

Platform SDK. По-долу е показан екран от инструмента показващ типовата библиотека на файла **comctrl32.ocx**, който се разпространява с Windows.

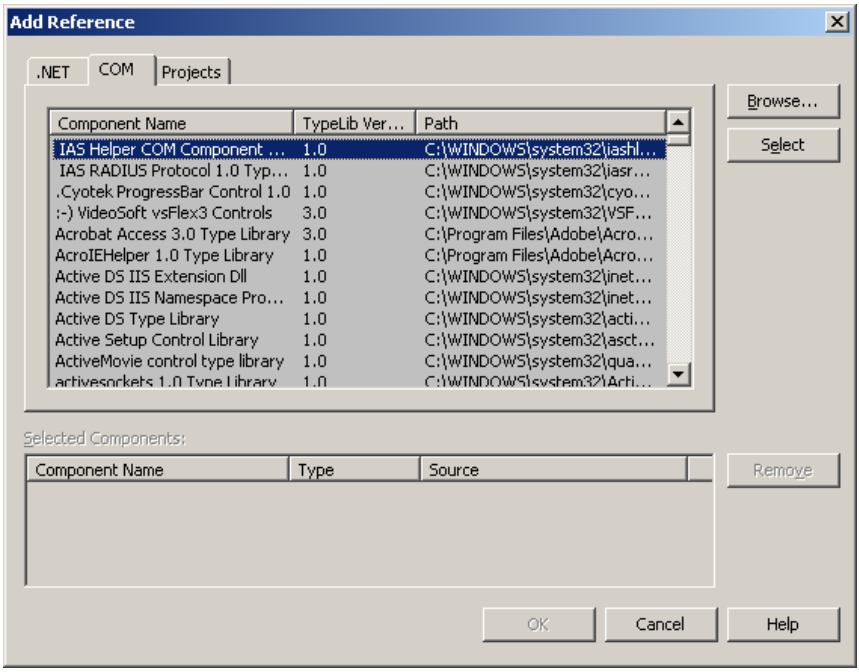


За да се осъществи извикването на даден COM обект, управлявания код трябва да получи информацията от типовата библиотека и да разбере какви методи и класове има този COM обект. Този проблем е решен с въвеждането на Interop асемблы. То съдържа всичката информация, която има в една типова библиотека, представена като класове и типове в управляван код, така че програмиста да може съвсем свободно да се обръща към COM обекта, както към всеки един обект от управлявания код.

Нека сега да разгледаме какви са начините за създаване на тази Interop асемблита.

Генериране на Interop асемблы чрез Visual Studio .NET

Средата за разработка Visual Studio .NET предлага много лесен начин за извикване на COM обект като се грижи за генерирането на Interop асемблито. Всичко, което трябва да направите е да отидете на References на вашия проект, да изберете Add Reference и след това от етикета COM да изберете COM обекта, който желаете. Имайте предвид, че в този списък са показани само регистрираните в системата COM обекти. За справка, регистрирането на COM обектите е описано в секцията "Видове COM обекти и регистрация". Фигурата по-долу показва как изглежда прозорецът за избиране на COM обект:



След като изберете COM обекта, в папката, където се компилира вашето приложение ще се появи Interop асемблито. Възможно е да се появи повече от едно асемблии, ако COM обекта, който сте избрали е свързан с друг COM обект. Тогава VS.NET ще генерира Interop асемблии и за другия COM обект. Имената на генерираните от VS.NET Interop асемблита започват с Interop. следван от името на обекта.

Възможно е VS.NET да не генерира Interop асемблии. Това става, в случай че за избрания COM обект има инсталирано Primary Interop Assembly (PIA). Primary Interop Assembly се генерира от производителя на съответния COM обект и се инсталира на вашата машина обикновено като част от по-голямо приложение. Например Microsoft Word и Microsoft Excel имат PIA, които се инсталират заедно с Microsoft Office пакета след като бъде избрана опцията ".NET Programmability Support". Тези PIA се инсталират в GAC (Global Assembly Cache), с което се осигурява поддръжката на различни версии и защита от неупълномощено подменяне. Ако все пак решите, че е необходимо да копирате PIA локално при вашето приложение, това може да стане като изберете опцията "Copy Local" на **`true`** от свойствата на добавения COM обект.

Генериране на Interop асемблии чрез tlbimp.exe

В случай, че не използвате средата Visual Studio, генерирането на Interop асемблии може да стане с помощта на инструмента **`tlbimp.exe`**. Задължителен параметър при извикването на **`tlbimp`** е пътят, където се намира описанието на библиотеката (tlb описанието). Това може да е самия COM обект или отделен **`.tlb`** файл. Полезен параметър при извикването на **`tlbimp`** е **`/namespace`**. С този параметър се задава какво да е простран-

ството от имена на генерираното асембли. Това е помага понякога, когато имате конфликт с автоматично генерираното пространство от имена и пространствата използвани във вашия проект или пък просто да изберете по-кратко и по-подходящо пространство от имена на Interop асемблито. **Tlbimp** има още доста опции, на които няма да се спираме тук, но е добре да се запознаете с тях, ако решите да използвате този инструмент. Името му, често се бърка с противоположния **tlbexp**, затова запомнете, че ролята на tlbimp е да импортира **.tlb** описание в зададеното асембли.

Програмно генериране на Interop асембли

Ако някога ви се наложи да генерирате собствено Interop асембли по време на изпълнение на програмата ви, класа **System.Runtime.InteropServices.TypeLibConverter** определено ще ви е от полза. Общо взето горните два метода работят точно с този клас, а използването му е повече от лесно. За да генерирате едно Interop асембли е необходимо да извикате метода **TypeLibConverter.ConvertTypeLibToAssembly(…)** като му подадете обекта, който съдържа **.tlb** описанието.

Разпространение (deployment) на Interop асембли

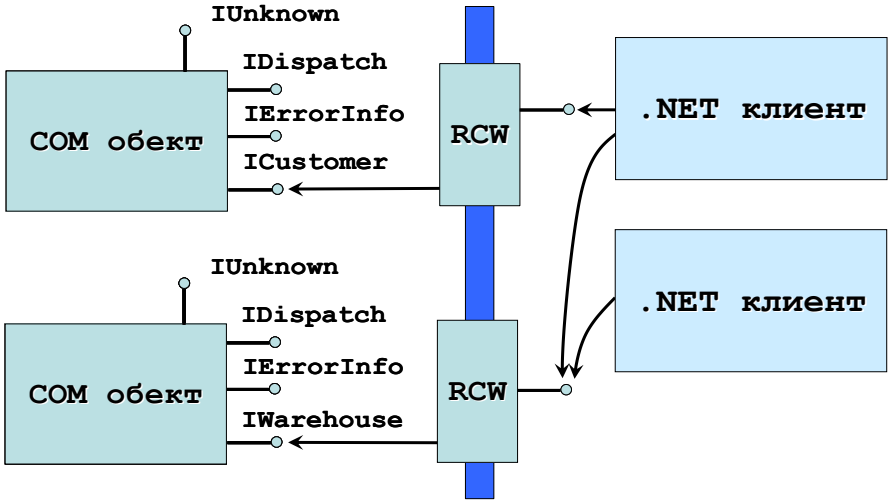
Разпространението на Interop асембли не се различава по нищо от разпространението на всяко друго асембли от .NET Framework. Достатъчно е да копирате асемблито в желаната директория. Имайте предвид обаче, че ако инсталирате програмата ви на друг компютър, трябва да се уверите, че COM обекта, който използвате е регистриран на този компютър. В противен случай вашето Interop асембли няма да намери COM обекта, за който се отнася, и няма да работи. С други думи, при подготвянето на инсталацията на вашата програма, трябва да включите и всички използвани COM обекти и да ги регистрирате на приемната машина.

Runtime Callable Wrapper (RCW)

Как всъщност става цялата магия с извикването на COM обектите? Какво толкова има в това Interop асембли? Представено по-прост начин, Interop асемблито съдържа информацията, която описва един COM обект, но представена във формат разбираем от .NET Framework. Това е целта на преобразуването и създаването на Interop асембли. По нататък, за да осъществи извикването на COM обекта, CLR използва тази информация и динамично създава Runtime Callable Wrapper (RCW). CLR създава точно един RCW за всеки COM обект, като по този начин RCW играе ролята на мост към даден COM обект. Както споменахме по-горе, инфраструктурата на COM изисква всяка една инстанция на COM обект да се отброява. RCW се грижи за това, както и за извикването на други стандартни интерфейси от инфраструктурата на COM като **IDispatch**, **IErrorInfo** и др. без програмиста да се обременява с тяхното съществуване. И не на последно място RCW извършва стандартното преобразуване на типовете от неуправлявана към управлявана среда.

Извикване на COM обект чрез RCW

Показаната по-долу фигура представя съвсем просто как се извършва едно извикване на COM обект:



Клиентът, използващ управляван код, се обръща към интерфейса **`ICustomer`** или **`IWarehouse`** на съответния COM обект. Извикването минава задължително през RCW, който от своя страна извиква интерфейсът **`IUnknown`**, за да определи дали търсения интерфейс се имплементира от този COM обект и в случай че го намери, извиква метода **`AddRef`** на **`IUnknown`**, за да укаже наличието на нова връзка.

Изниква въпросът как RCW разбира кога съответния му COM обект е свободен и няма повече връзки към него.

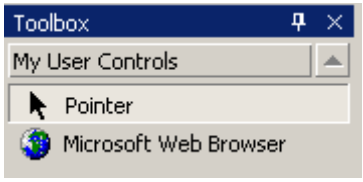
Поради липсата на детерминистични деструктори в .NET Framework връзката към COM обекта ще бъде освободена едва при извикването на системата за почистване на паметта. За да ускорим този процес може да използваме методът `Marshal.ReleaseComObject`. Този метод намалява брояча на връзките, който RCW поддържа за подадения COM обект. При достигане на брояча до нула, всички връзки с неуправлявания код се освобождават и обекта се унищожава. По нататъшното извикване на този обект ще доведе до изключение `System.NullReferenceException`.

Извикване на COM обект от управляван код – пример

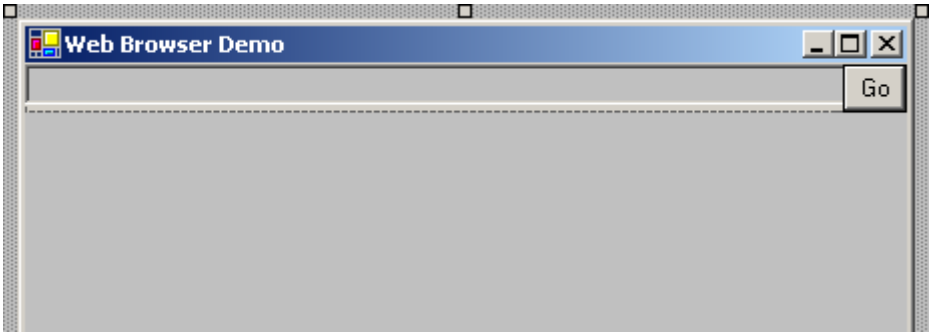
Ще разгледаме как с помощта на VS.NET може да добавим готов COM обект към управляван проект, да достъпваме неговите свойства и да използваме предоставената от него функционалност.

1. Стартираме VS.NET и създаваме нов Window Forms проект.

2. Отваряме Toolbox и с десен бутон избираме Add/Remove Items…
3. Избираме етикета COM components и от списъка избираме Microsoft Web Browser, маркираме го като избран, след което натискаме OK.
4. Избраният компонент се появява в панела Toolbox под името Microsoft Web Browser:



5. Отваряме формата на новосъздадения проект в режим на дизайн, хващаме компонента Microsoft Web Browser от ToolBox и го довличаме върху формата. На свойството Name присвояваме стойност **`axWebBrowser`**.
6. Над компонента Web Browser добавяме текстово поле с Name - **`textBoxUrl`** и бутон с надпис Go и Name - **`buttonGo`**, така че формата да добие следния вид.



7. Щракваме два пъти върху бутона и променяме функцията **`buttonGo_Click`** по следния начин:

```
private void buttonGo_Click(object sender, System.EventArgs e)
{
   string url = textBoxUrl.Text;
   axWebBrowser.Navigate(url);
}
```

8. Стартираме проекта с **[Ctrl+F5]**.
9. При набиране на уеб адрес в текстовото поле и натискане на бутона Go, страницата се зарежда в уеб браузър контролата.

Разкриване на .NET компонент като COM обект

След като разгледахме как един COM обект може да се използва от .NET компонент, ще разгледаме обратната задача. Как да се представи един

.NET компонент като COM обект, така че да може да бъде извикван от други COM обекти?

Преди всичко трябва да са изпълнени изискванията, които се налагат за всеки един COM обект. Това са наличието на уникален GUID, на ProgId идентификатор и регистрация в Windows Registry. За наше улеснение всички тези неща стават много лесно чрез използване на атрибути. По-долу е показано как чрез атрибута **[GuidAttribute]** се извършва описанието на необходимите параметри:

```
[Guid("D069E57A-981F-4841-8D68-E2F2342E92A2"),
    ProgId("SomeApplication.SomeClass")]
public class SomeClass
{
    // …
}
```

Инструментът regasm

Регистрацията на така обозначения клас може да се извърши ръчно чрез използването на инструмента **regasm.exe** или с VS.NET.

Ако решите да използвате **regasm.exe** имайте предвид, че освен за регистрация, той може да се използва и за изтриване на регистрацията. RegAsm приема като входен параметър пътя към асемблито, което трябва да се регистрира като COM обект. С опцията **/tlb** може да укажете да се генерира и регистрира типовата библиотека, която този .NET компонент представлява. Ако зададете опцията **/unregister**, регистрацията на вашия .NET компонент ще бъде премахната и той ще спре да бъде разпознаван като COM обект. Допълнителна опция е **/regfile**. Тя позволява генерирането на файл с команди за Windows Registry, необходими за регистрацията на .NET компонента като COM обект. Имайте предвид, че опциите **/tlb** и **/regfile** са взаимно изключващи се. Използването на опцията **/regfile** не извършва регистрацията в Windows Registry.

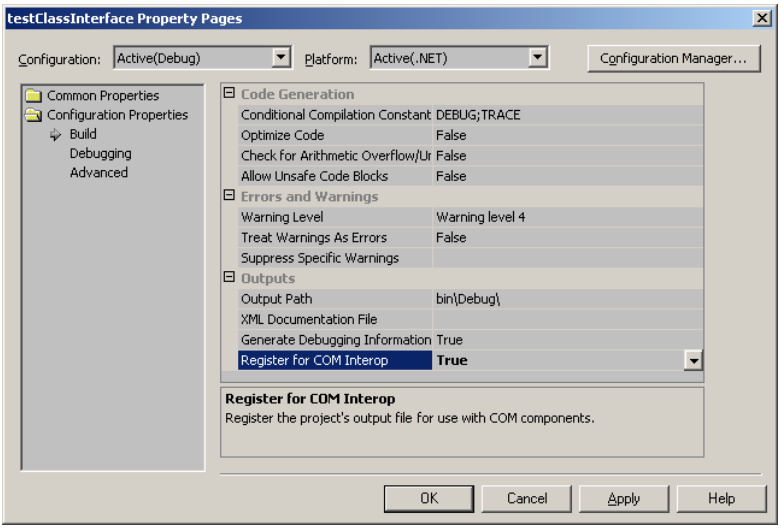
Освен **regasm** има още един инструмент за генериране на типова библиотека. Това е **tlbexp** и както самото име говори, този инструмент експортира типовата библиотека на .NET компонент. Сама по себе си генерираната типова библиотека не е достатъчна за работата на взаимодействието на .NET с COM, затова този инструмент не се използва често. Еквивалентната работа, включително и регистрацията на COM обекта се извършва от **regasm**.

Атрибути за регистрация и дерегистрация

Допълнителен контрол върху регистрацията може да получите с помощта на атрибутите **[ComRegisterFunction()]** и **[ComUnregisterFunction()]**. Те се прилагат върху методи и се изпълняват съответно в момента на регистрация и дерегистрация. С тях програмистът има възможност да

направи допълнителни инициализации, като например да добави нови ключове в Windows Registry или да създаде временни ресурси.

Разбира се, съществува и вариант за регистрация на .NET компонент от VS.NET. Този вариант, обаче, е подходящ само за целите на разработката тъй като изисква наличието на VS.NET. По-долу е показана опцията Register for COM Interop във VS.NET. До тази опция може да стигнете като отворите свойствата на вашия проект и изберете настройките за Configuration Properties\Build. Имайте предвид, че тази опция не е активна за всички типове проекти като Windows Forms, Console Application или др., за които регистрацията като COM обект няма конкретен смисъл.



За допълнителен контрол върху типовата библиотека .NET Framework предоставя набор от атрибути, които се прилагат върху .NET класа. Това са **`CoClassAttribute`**, **`ComVisibleAttribute`**, **`GuidAttribute`** и др. Няма да се спираме на тях, тъй като се изискват по-сериозни познания по COM и излизат извън целта на тази книга.

COM Callable Wrapper (CCW)

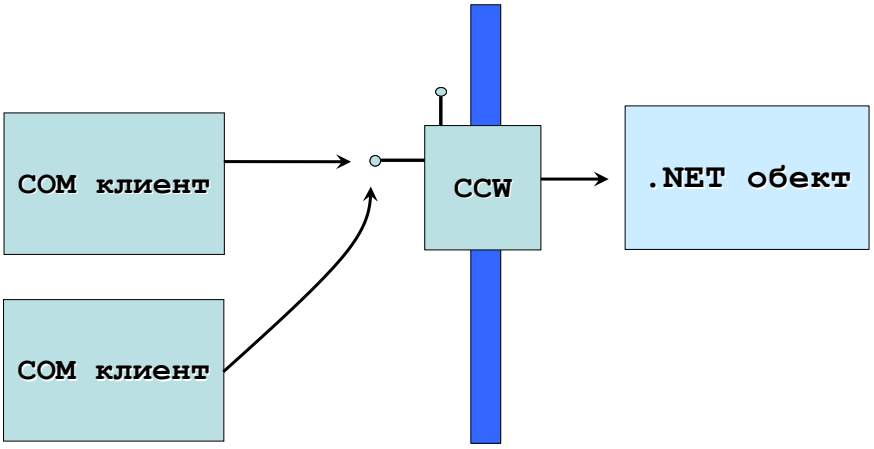
Аналогично на COM обектите, .NET компонентите също се достъпват през специално прокси, чиято грижа е да осъществи преобразуването на данните. Това прокси се нарича COM Callable Wrapper и се грижи още за поддръжката на сигурността и автоматичното почистване на паметта, които не съществуват в инфраструктурата на COM и не могат да бъдат използвани от COM. Тук също важи правилото, че за всеки един .NET компонент съответства точно един CCW.

Прокси класът трябва да осигури и имплементацията на стандартните интерфейси за COM – **`IUnknown`**, **`IDispatch`**. Само така .NET компонента отговаря на изискванията на COM инфраструктурата и може да се представи като COM обект. Една препоръка, която ще спомена по-надолу отново и която трябва да запомните, е, че за да си осигурите безпроб-

лемно извикване на .NET компонент трябва да използвате интерфейси, които дефинират поведението на вашия управляван клас и които се имплементират от него. Така ще се приближите максимално близо до начина, по-който се представят COM обектите и ще се спестите проблеми при управление на версиите и генериране на типова библиотека.

Извикване на .NET компонент чрез CCW

От фигурата по-долу се вижда схематично процеса на извикване на един .NET компонент:



COM клиентите се обръщат към CCW, който имплементира стандартните **`IUnknown`** и **`IDispatch`** интерфейси и преобразува данните към .NET компонента (двете кръгчета, които излизат от CCW са обозначение за интерфейс, когато става въпрос за COM обект).

Изисквания към .NET типове за ползване от COM

В описанието на CCW споменахме една препоръка за осигуряване на безпроблемно извикване на .NET компонент. Истината е, че за да извикате един .NET компонент от COM е нужно да се спазят доста условия тъй като технологията COM е по-стара от .NET и има своите ограничения. По-долу са изброени условията, които трябва да бъдат изпълнени, за да се осигури достъп от COM обект:

- Всички управлявани класове трябва да бъдат **`public`**.
- Всички управлявани класове трябва да имат публичен конструктор по подразбиране (конструктор без параметри).
- Методи, свойства, полета и събития трябва да бъдат **`public`**.
- Класовете не могат да бъдат абстрактни.
- Препоръчва се класовете да имплементират интерфейс.
- Избягвайте статични методи.

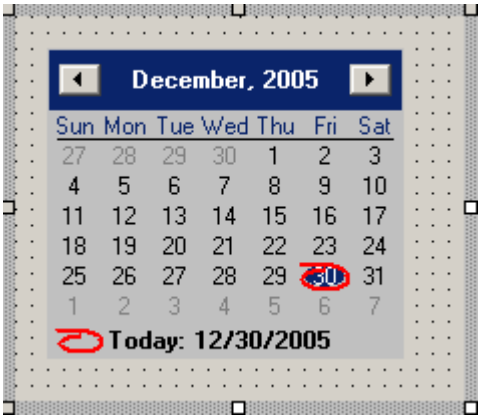
Разкриване на .NET компонент като COM обект – пример

В този пример ще направим .NET потребителска контрола и ще я регистрираме като COM компонент. Ще използваме Internet Explorer за да визуализираме контролата като ActiveX компонент.

1. Отваряме VS.NET.
2. Създаваме нов Class Library проект.
3. Създаваме нов User Control с име **`CalendarControl`**.
4. Върху декларацията на създадения клас **`CalendarControl`** прилагаме следните атрибути:

```
[GuidAttribute("D069E57A-981F-4841-8D68-E2F2342E92A2"),
  ProgId("Demo_4_RegisterAsCOM.TimeControl")]
```

5. Превключваме в режим на дизайн и довличаме Calendar контрол върху създадения **`CalendarControl`**.



6. Създаваме нов HTML файл **`TestCalendarControl.html`**.
7. В **`<BODY>`** тага поставяме следния **`<OBJECT>`** таг, за да извикаме потребителския контрол:

```
<object classid="CLSID:D069E57A-981F-4841-8D68-E2F2342E92A2">
</object>
```

8. Забележете, че в **`<OBJECT>`** тага задаваме същия GUID, който използвахме при указването на атрибута **`GuidAttribute`** на класа.
9. Последната стъпка е да укажем на VS.NET, че е необходимо да регистрира нашия контрол като COM обект.
10. Избираме Properties на текущия проект и от Configuration Properties / Build настройваме Register for COM Interop да има стойност True.
11. Компилираме приложението.

12. Щракваме с десен бутон върху **TestCalendarControl.html** и избираме View in Browser.
13. В отворения прозорец се визуализира, подобно на ActiveX компонент, нашият контрол с вграден календар от управляваната библиотека.

Взаимодействие със C++ чрез IJW

Повечето закоравели (hardcore) C++ писачи навярно вече са се запитали, как може да се извика .NET клас от C++ среда. Това всъщност въобще не е трудно. Използва се така наречената технология IJW (It Just Works), която позволява почти интуитивно извикването на един .NET клас. Аналогично на RCW и CCW при IJW също има прокси, което отговаря за преобразуването на типовете от неуправлявана към управлявана среда и обратно. За този тип прокси е съществено, обаче, че е с висока производителност. Измерено е, че IJW проксито води до около 10-30 машинни инструкции за всяко извикване на управляван код.

Използването на библиотека от .NET средата става с помощта на **using** директивата. Да разгледаме следния пример.

IJW извикване от C++ – пример

Ето един кратък пример, който илюстрира използването на IJW технологията от C++:

```
#using <mscorlib.dll>
#include <stdio.h>
#include <iostream>

using namespace std;

void main()
{
    // Declare unmanaged pointer of type char*
    const char* str = "IJW (It Just Works)";

    // Call unmanaged function "printf"
    printf("%s\n", str);

    // Call unmanaged function "ostream::operator <<"
    cout << str << endl;

    // Call managed function "Console::WriteLine"
    System::Console::WriteLine(str);
}
```

На първия ред в примера е показано как се използва директивата using за достъп до .NET средата и по конкретно до файла **mscorlib.dll**, който е

входната точка при стартирането на всяко едно .NET приложение. Следва деклариране на символен низ и подаването му към **System::Console::WriteLine()** метода. Виждате, че никъде не е се извършва ръчно преобразуване на типовете и въпреки това извикването на управлявания метод е успешно. Нуждата от използване на атрибути също се елиминира за разлика от Platform Invoke. Все пак имайте предвид, че когато използвате IJW ще ви се налага много по-често ръчно да указвате начина на преобразуването на променливите с помощта на класа **Marshall**. Поради тази причина използването на IJW, може доста да усложни разработката на вашето приложение и използването на IJW се препоръчва само когато се търси значително подобрение в бързината за изпълнение. Най-подходящ IJW е също в приложения, които ползват активно неуправляван код и където е необходимо да се открие и разреши проблем с производителността.

Препоръки за използване на .NET типове от COM

По-долу ще посочим някои препоръки за подобряване на скоростта на изпълнение, за преобразуване на .NET типове в неуправлявана среда и за извикване на методи на неуправлявана от управлявана среда и обратно.

- Използвайте "chunky" вместо "chatty" интерфейси. Имената на този тип интерфейси показват начина на предаване на информацията. За "chunky" интерфейсите е характерно, че те с помощта на малък брой методи се извършва голяма част от работата. Това означава по-малък брой преминавания от едната среда към другата и естествено по-малък брой преобразувания на типовете и по-добра производителност. При "chatty" интерфейсите работата се извършва с помощта на серия от извиквания на голям брой методи и необходимата информация се извиква или предава на части. Този тип комуникация не се препоръчва, когато е налице преминаване от една среда към друга поради значителното време необходимо за преобразуване на типовете.
- Имплементирайте **IDisposable** за неуправляваните ресурси. Всеки неуправляван ресурс, който използвате, е добре да бъде капсулиран в управляван клас. Управляваният клас трябва да имплементира **IDisposable**, в случай че възникне изключение неуправлявания ресурс да бъде освободен.
- Избягвайте късно свързване. Късното свързване (late binding) е техника, която позволява един COM обект да бъде извикан по време на изпълнение на програма, без да е имало информация за неговите типове и методи по време на компилацията на програмата. Пример за това е използването на интерфейса **IDispatch**. При късното свързване не е нужно да има създадено Interop асемблы, но за сметка на това се използва отражение (reflection), което забавя значително извикването на COM обекта.

- Указвайте името на метода, който искате да извикате изрично чрез **DllImport**. Когато използвате атрибута **DllImport** използвайте свойството му **ExactSpelling** със стойност **true**, за да избегнете претърсването на всички методи за подобно име.
- Оптимизирайте преобразуването на данни. При преобразуването трябва да се вземе предвид, че използването на т.нар. "blittable" типове (типове по стойност, масиви от стойностни типове и структури) дава много висока производителност, тъй като те директно се копират, без да се извършва преобразуване.
- Може да използвате **SuppressUnmanagedCode** атрибута за критични по скорост извиквания. Атрибутът **SuppressUnmanagedCode** позволява да се елиминира обхождането на стека, за да се извърши необходимата проверка за права. Изисква се всички извикващи да имат дадено право **UnmanagedCode**, за да може да се извика неуправлявания код. Използвайте този атрибут само за критични по скорост извиквания, тъй като употребата му представлява риск за сигурността на вашето приложение.
- Следете броячите за взаимодействие. Вградените в Windows броячи (performance counters) могат да ви дадат информация за броя на преобразуванията и създадените CCW. Броячите за взаимодействие се намират в категорията .NET CLR Interop.
- Използвайте CLR Spy [6] за да откриете евентуални проблеми. Инструментът CLR Spy е задължителен за всеки, който смята да прави нещо по-сериозно в извикването на неуправляван код. Този инструмент съдържа набор от т.нар. проби, който могат да засичат неправилно форматирани P/Invoke извиквания, ненавременни извиквания на системата за управление на боклука и др.

Immutable ли са наистина символните низове?

Ще разгледаме малък пример, с който се показва нуждата от програма като CLR Spy и междувременно ще проверим наистина ли са immutable символните низове в .NET Framework. Примерът и описанието са базирани на публикацията на Chris Brumme – "Interning Strings & immutability" [7].

Примерната програма по-долу използва P/Invoke, за да извлече името на компютъра. За "късмет" обаче, програмистът е използвал променливата **computerName** от тип **string**, вместо да използва типа **StringBuilder**, който е задължителен, когато неуправлявания код връща низ с променлива дължина:

```
using System;
using System.Runtime.InteropServices;

public class GetComputerNameDemo
{
```

```
    static void Main(string[] args)
    {
        String computerName = "strings are always immutable";
        String otherString = "strings are always immutable";

        int len = computerName.Length;
        GetComputerName(computerName, ref len);

        Console.WriteLine(otherString);
    }

    [DllImport("kernel32", CharSet=CharSet.Unicode)]
    static extern bool GetComputerName(
        [MarshalAs (UnmanagedType.LPWStr)] string name,
        ref int len);
}
```

За още по-голям "късмет" символните низове **computerName** и **otherString** са интернирани поради еднаквото си съдържание. Извикването на функцията **GetComputerName()** води до промяна на **computerName**, но заедно с него се променя и **otherString**. Поради факта, че промяната на низа става в неуправляван код, .NET Framework няма никакъв начин да разбере, че някои е направил тази промяна. В резултата променяме съдържанието не на един, а на два низа, без дори да създаваме нова инстанция на класа, което е в огромно противоречие на идеологията за неизменимите (immutable) низове.

Използването на инструмента CLR Spy намаля възможността от такива грешки поради наличието на Customer Debug Probes (CDP). Това е нова функционалност в CLR, която ни дава възможност да откриваме често срещани грешки и за наше щастие тези CDP са насочени предимно към откриване на грешки при Interop и преобразуване.

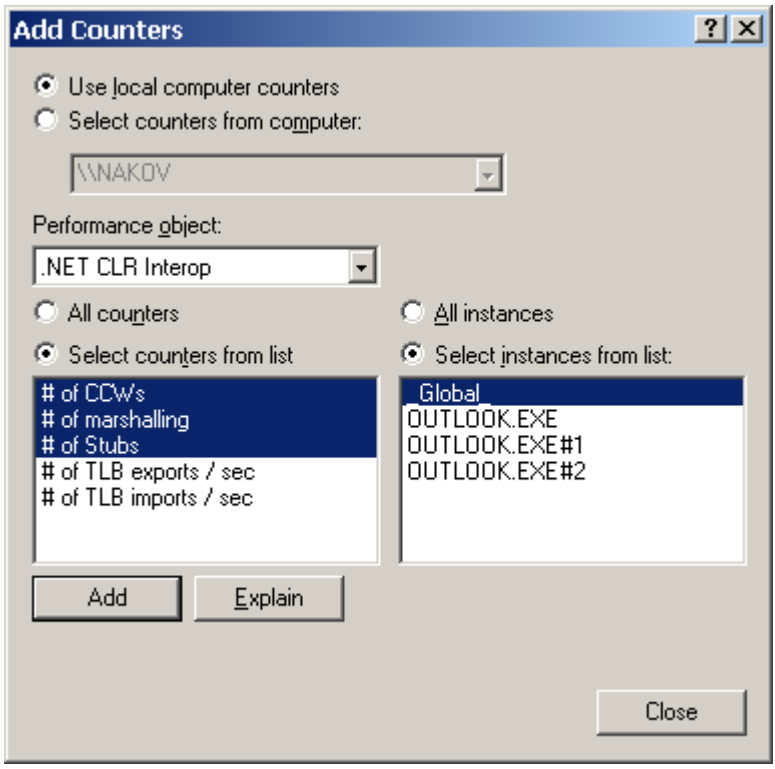
Използване на броячи за производителност и CLRSpy – пример

В този пример ще разгледаме как се използват броячите за производителност и как да използваме CLRSpy за наблюдение на проблемни места при взаимодействието с неуправляван код.

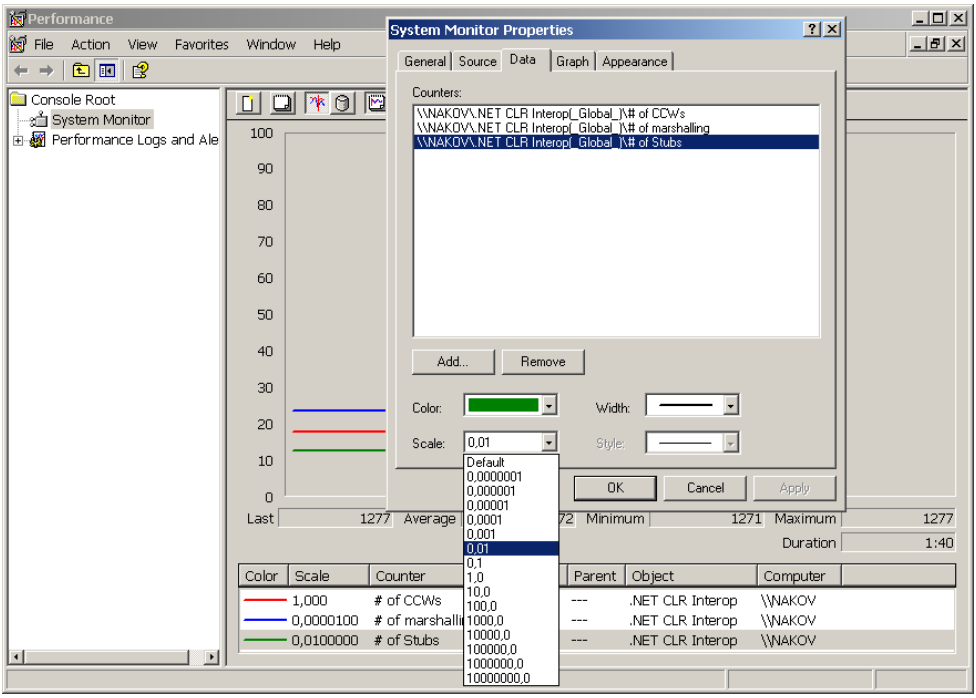
1. Стартираме инструмента за наблюдение на Windows броячите "Performance Monitor" (**perfmon.exe**):

```
Start → Settings → Control Panel → Administrative Tools → Performance
```

2. Създаваме нов набор от броячи (New Counter Set).
3. Добавяме в набора броячите за .NET CLR Interop:

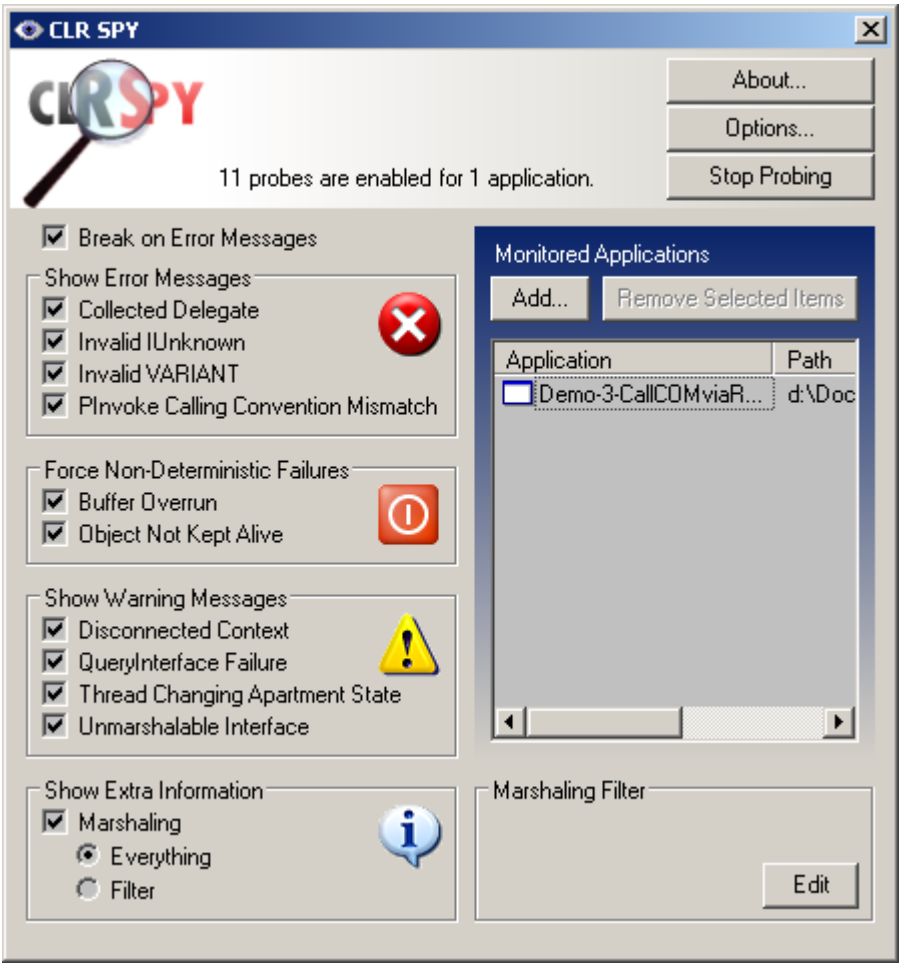


4. Задаваме подходящ цвят и скала за всеки от броячите, така че графиката да се събира в полето, което е отделено за нея:



5. Стартираме проекта от демонстрация #3 (**Demo-3-CallCOMviaRCW.exe**) и проследяваме промяната в графиката на броячите.

6. Стартираме инструмента CLRSpy. Добавяме към списъка на наблюдаемите приложения **Demo-3-CallCOMviaRCW.exe**. Стартираме приложението **Demo-3-CallCOMviaRCW.exe** и наблюдаваме отчетените от CLRSpy събития:



7. Отваряме файла **C:\clrspy.log** и разглеждаме всички отчетени събития:

```
...
[12/30/2005 3:17:40 PM] Marshaling in Demo-3-CallCOMviaRCW.exe
(PID 2528): Marshaling from Int32 to DWORD in method SetExtent.
[12/30/2005 3:17:40 PM] Marshaling in Demo-3-CallCOMviaRCW.exe
(PID 2528): Marshaling from IntPtr to DWORD in method GetWindow.
```

```
[12/30/2005 3:17:45 PM] Marshaling in Demo-3-CallCOMviaRCW.exe
(PID 2528): Marshaling from Int32 to DWORD in method
OnInPlaceDeactivate.

[12/30/2005 3:17:45 PM] Marshaling in Demo-3-CallCOMviaRCW.exe
(PID 2528): Marshaling from Int32 to DWORD in method Unadvise.

[12/30/2005 3:17:45 PM] Marshaling in Demo-3-CallCOMviaRCW.exe
(PID 2528): Marshaling from Int32 to DWORD in method Unadvise.
```

Упражнения

1. Имплементирайте Windows Forms приложение, което показва списък с активните в момента процеси. За всеки процес трябва да се покаже следната информация: идентификатора му (PID), името на файла, от който е зареден, приоритета му, обема на минималната и максималната му работна памет (working set). Използвайте API функциите **`EnumProcesses()`**, **`OpenProcess()`**, **`GetModuleBaseName()`**, **`GetPriorityClass()`**, **`GetProcessWorkingSetSize()`** и **`CloseHandle()`**, като ги извиквате през P/Invoke. Дефинициите са в библиотеките **`kernel32.dll`** и **`psapi.dll`**. Използвайте документацията и примерите от MSDN за да видите как се използват посочените функции. Визуализирайте по подходящ начин извлечената информация за процесите.
2. Имплементирайте Windows Forms приложение, което визуализира PDF документи с помощта на COM компонента "Adobe Acrobat Control for ActiveX".
3. Създайте Windows Forms контрол, който реализира играта "морски шах". Направете контролът достъпен като COM сървър. Направете HTML страница, с която да визуализирате контрола в Internet Explorer.
4. Реализирайте конзолно приложение, което по даден XML файл, съдържащ списък от фирми и информация за тях, генерира MS Excel документ, съдържащ същата информация във вид на таблица. Всяка фирма се описва с име, адрес и телефон. За връзка с MS Excel използвайте COM компонентата "Microsoft Office Spreadsheet".

Използвана литература

1. Мартин Кулов, Взаимодействие с неуправляван код – http://www.nakov.com/dotnet/lectures/Lecture-22-Interoperability-v1.0.ppt
2. MSDN Library – http://msdn.microsoft.com
   - Interoperating with Unmanaged Code
   - An Overview of Managed/Unmanaged Code Interoperability
   - Beyond (COM) Add Reference: Has Anyone Seen the Bridge?
   - Using the .NET Framework SDK Interoperability Tools

- Calling a .NET Component from a COM Component
- Microsoft Office and .NET Interoperability
- The Myth of .NET Purity, Reloaded
- Platform Invocation Services

3. MSDN Magazine – http://msdn.microsoft.com/msdnmag/
   - Calling Win32 DLLs in C# with P/Invoke
   - Migrating Native Code to the .NET CLR
4. Improving .NET Application Performance and Scalability (MS Patterns and Practices) – http://msdn.microsoft.com/library/en-us/dnpag/html/scalenet.asp
   - Chapter 7 – Improving Interop Performance
   - Checklist: Interop Performance
5. P/Invoke .NET: The Interop wiki! – http://www.pinvoke.net/
6. Microsoft .NET/COM Migration and Interoperability – http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/cominterop.asp
7. CLR Spy – http://www.gotdotnet.com/Community/UserSamples/Details.aspx?SampleGuid=C7B955C7-231A-406C-9FA5-AD09EF3BB37F
8. Interning Strings & immutability - http://blogs.msdn.com/cbrumme/archive/2003/04/22/51371.aspx

Глава 24. Управление на паметта и ресурсите

Автори

Стоян Дамов

Димитър Бонев

Необходими знания

- Базови познания за .NET Framework и CLR
- Базови познания за общата система от типове в .NET (Common Type System)
- Базови познания за езика C#
- Незадължителни базови познания за езиците C и C++

Приключение. Ха-ха! Силни усещания. Ха-ха!
Джедаят не копнее за такива неща. ***Йода***

Съдържание

- Управление на паметта при различните езици и платформи
- Управление на паметта в .NET Framework
- Как се заделя памет в .NET?
- Как работи системата за почистване на паметта?
- Поколения памет
- Блок памет за големи обекти
- Финализацията на обекти в .NET
- Деструкторите в C#. Опашката Freachable
- Тъмната страна на финализацията
- Съживяване на обекти
- Ръчно управление на ресурсите с интерфейса **`IDisposable`**
- Базов клас, обвиващ неуправляван ресурс
- **`Close()`** и изрична имплементация на **`IDisposable`**
- Взаимодействие със системата за почистване на паметта
- Слаби референции
- Ефективно използване на паметта

- Техниката "пулинг на ресурси"

В тази тема...

В настоящата тема ще научите как да пишете правилен и ефективен код по отношение използването на паметта и ресурсите в .NET Framework. Ще започнем със сравнение на предимствата и недостатъците на ръчното и автоматично им управление. След това ще разгледаме по-обстойно автоматичното управление, фокусирайки се най-вече върху системата за почистване на паметта в .NET (т. нар. garbage collector). Ще обърнем по-голямо внимание на взаимодействието на вашия код с нея и практиките, с които можете да й помогнете да работи възможно най-ефективно.

Ако решите да прескочите тази тема, понеже на пръв поглед изглежда скучна, недейте! Тя е изпъстрена с примери за това какъв код да избягвате да пишете и какъв код да се стремите да пишете. В нея ще намерите имплементация на:

- Базов клас за многократна употреба, обвиващ неуправляван ресурс, който имплементира правилно интерфейса **`IDisposable`** и има финализатор, а също така е и безопасен за употреба в многонишкови програми (thread-safe)
- Клас за многократна употреба, имплементиращ thread-safe пул от ресурси

Управление на паметта при различните езици и платформи

В почти всички езици за програмиране, заделянето на динамична памет се извършва "ръчно", т.е. се обявява изрично от програмиста. Разликата е в освобождаването и съществуват три начина за освобождаване на памет[1]:

- **Ръчно** – паметта се освобождава изрично от програмиста, например в C и C++.
- **Автоматично** – паметта се освобождава от система за автоматично почистване на паметта (често наричана система за почистване на боклук, Garbage Collector, GC), която обикновено се задейства автоматично при недостиг на памет. Такъв подход се използва например в езици като Java, при които кодът се изпълнява от виртуална машина, или като при повечето .NET езици, където кодът се изпълнява в контролирана среда, осигуряваща коректно изпълнение на кода (.NET CLR).
- **Смесено** – паметта може да се освобождава както директно от програмиста, така и автоматично от система за почистване на боклук, например Visual Basic (версиите преди VB.NET).

Придобиването и освобождаването на ресурс, различен от памет (например ресурс предоставен от операционната система), обикновено е свързано с ръчно управление, въпреки че е възможно освобождаването да се автоматизира.

Първо ще разгледаме ръчното управление на памет, като посочим начините за заделяне и освобождаване на памет и ресурси в два известни езика от по-ниско ниво – C и C++. След това ще разгледаме автоматичното управление в .NET и ще сравним възможно най-обективно двата начина за управление на паметта и ресурсите за да имате ясна представа какво можете и какво не можете да направите във вашите настоящи и бъдещи .NET приложения.

Ръчно управление на паметта и ресурсите

Страхът е пътят към тъмната страна. Страхът води до използване на управлявани езици. Управляваните езици водят до използването на Garbage Collector. Garbage Collector води до страдания. ***C/C++ Йода***

В ранните години на компютърното програмиране паметта и резурсите се управляваха ръчно, чрез оператори и функции за заделяне и освобождаване на памет и ресурси. По-късно се появиха някои техники за автома-

[1] Като изключим напълно валидната стратегия на бездействие

тизация, които улесниха работата с паметта, а след време тези техники започнаха да се внедряват в езиците за програмиране и платформите за разработка и изпълнение на софтуер. Нека разгледаме управлението на паметта в езиците C и C++.

Управление на паметта в езика C

Заделянето и освобождаването на памет в езика C се прави ръчно от програмиста посредством библиотечните функции **malloc(…)**, **realloc(…)** и **free(…)**[2]. Функцията **malloc(…)** заделя блок последователни байтове от динамичната памет (т. нар. heap) и връща указател към първия байт от тази памет, **free(…)** я освобождава, а **realloc(…)** може да заделя, освобождава, разширява и премества блокове памет и е своеобразен менажер на паметта (memory manager), чрез който могат да се имплементират **malloc(…)** и **free(…)**.

Управление на паметта в езика C++

В езика C++, освен гореизброените функции можете (и се препоръчва) да използвате вградените в езика двойка оператори **new** и **delete**[3] (които в повечето случаи са имплементирани посредством **malloc(…)** и **free(…)**). Предимството на оператора **new** пред функцията **malloc(…)** е, че след като задели памет за инстанция на даден тип, операторът извиква код за инициализация на типа (наричан конструктор) в тази памет. Операторът **delete** извиква код за разрушаване на инстанцията (наричан деструктор), след което освобождава паметта, заета от оператора **new**[4].

Деструкторите в C++

Деструкторът на инстанция на определен тип (обект) се изпълнява, когато:

- се извика ръчно в кода;
- обектът напусне обхвата (scope), в който е създаден;
- или при възникване на изключение.

Създаване на обекти в C++

Създаването на обект в динамичната памет с малки изключения[5] става с помощта на вградения оператор **new**. В най-общи линии, той се опитва да

---

[2] Съществуват и други функции, като **calloc(…)**, а също и нестандартни, като **alloca(…)**, която заделя памет от стека

[3] При използване на масиви се използват операторите **new[]** и **delete[]**

[4] За примитивните типове като **int** не се извиква конструктор/деструктор

[5] С помощта на оператора "placement **new**" може да създадете обект на произволен адрес в паметта, включително и в стека

задели памет, достатъчна за да помести инстанция на подадения тип, след което, ако успее, извиква конструктора за да инициализира обекта в тази памет. Ако не успее, в зависимост от няколко условия, които няма да разглеждаме, или изхвърля изключение или записва нулева (**NULL**) стойност в указателя. Обект, заделен в динамичната памет, се разрушава с вградения оператор **delete**, който извиква деструктора на обекта и ако не възникне изключение, освобождава заделената памет от оператора **new**.

Автоматично унищожаване на ресурси в C++

Фактът, че деструкторът на обект, инстанциран в стека, се извиква автоматично при напускането на неговия обхват или при възникване на изключение (и в частност техниката RAII[6]), е може би най-важната причина C++ да не се нуждае от клаузата **finally**, без която не е възможно да се пише код, устойчив на изключения, в езици като C# и Java[7]. Наличието на деструктори в C++ прави възможно автоматичното освобождаване на всички видове ресурси, например:

```
// след напускане на обхвата на mem паметта ще бъде освободена
boost::shared_ptr mem(new char[20]);
// дори при възникване на изключение тук паметта ще се освободи,
// защото деструкторът на shared_ptr ще бъде извикан
```

Освобождаването на паметта и ресурсите в C е възможно да се прави само ръчно (освен когато се ползва Garbage Collector за C).

Предимства и недостатъци на ръчното управление на паметта и ресурсите

Ще изброим предимствата и недостатъците, тъй като представянето им в табличен вид е неудобно за четене.

Предимства на ръчното управление

Предимствата са повече от изброените по-долу, но няма да даваме пълен списък, тъй като се фокусираме върху .NET:

- Възможно е освобождаване на ресурсите в известен, желан момент (например извикването на **free** или **delete** или автоматичното извикване на деструктор в C++ ръчно, при излизане на обекта извън обхват или при възникване на изключение).
- Имаме пълен контрол на начина за заделяне и освобождаване на памет, включително написването и замяната на мениджъра на

[6] Resource Acquisition Is Initialization

[7] Подробно обяснение ще намерите в края на тази тема

паметта, както и предефинирането на заделяне и освобождаване на памет за желан от нас потребителски тип (в C++).

- Възможно е конструиране на обект на зададен от нас адрес (полезно в C++ за писане на устойчив на изключения код, както и в C при писане на системен код).
- Възможно е заделяне на блок памет от стека (с помощта на **`alloca(…)`** или използването на масиви с променлив размер - C99).
- Липсата на памет може да се установи по няколко начина и да се предприеме някакво действие.

Недостатъци на ръчното управление

Недостатъците са повече от споменатите по-долу и се надяваме, че изброените проблеми ще ви дадат достатъчно основание да разберете значимостта на системата за автоматично почистване на паметта (GC) в .NET Framework. Да започнем с най-често срещаните грешки, които водят до проблеми и се допускат дори от най-опитните програмисти:

- Несъответствие в броя на заделянията и освобождаванията, което води до "изтичане" на памет (memory leak). Недостатъкът очевидно е ръчното освобождаване на памет. Като частен случай трябва да посочим и изтичане на памет, породено от недобре написан конструктор на клас по отношение на възникването на изключения.
- Несъответствие в извикването на операторите за типове и масиви от типове, например извикване на **`delete`**, за памет, заделена с **`new[]`**.
- Опит за четене или писане на вече освободена памет или опит за повторно освобождаване на памет.
- Опит за писане в незаделена от програмиста памет на валиден адрес в адресното пространство на вашата програма или запис на повече информация от заделената за това памет – проблем, допринесъл за най-големите пробиви свързани със сигурността.

Не можем да не бъдем честни към C/C++ програмистите и да споменем, че за повечето от горепосочените проблеми съществува решение – вдигане на нивото на предупреждения от компилатора, използването на assertions, т. нар. умни указатели (smart pointers), STL контейнери, мощни библиотеки като boost и техниката RAII.

Ето и няколко недостатъка, за които също съществуват решения, но в повечето случаи те са свързани с допълнителни разходи:

- Бавно заделяне (и освобождаване) на динамична памет, особено с мениджъра на паметта по подразбиране.
- Фрагментиране на динамичната памет поради неоптимизирана реализация на мениджъра на паметта по подразбиране.

- Неефективно използване на процесорите на машината поради неоптимизирани алгоритми за синхронизация на структурите от данни на мениджъра на паметта по подразбиране (т. нар. false sharing, при който всички процесори блокират докато един от тях изпълнява код, заделящ или освобождаващ памет)[8].
- Цяла глава може да се посвети на една от най-трудните и податливи на грешки задачи в C++ - броене на референциите към обектите, най-вече при йерархии от обекти, където има циклични референции. При такива проблеми, написването на правилен код с броене на референции граничи с героизъм.

Трябва да отбележим, че съществуват свободни (при това доста добри) имплементации на Garbage Collector за C и C++, които обаче не са широко разпространени и използвани.

Изброените недостатъци по-горе са довели до създаването на скъпи продукти като Insure++, Rational Purify и CompuWare BoundsChecker, които да се справят с куп проблеми, които както ще се убедите сами, просто не съществуват в .NET.

Управление на паметта в .NET Framework

Трябва да отучиш това, което си научил. ***Йода***

В секциите до края на тази глава ще разгледаме особеностите на управлението на паметта в .NET Framework. Ще се спрем на процесите, протичащи зад сцената на автоматичното управление на паметта. Ще проследим жизнения цикъл на обектите – от заделянето на памет при тяхното създаване, до момента в който те умират и освобождават заетите от тях ресурси. Ще спомена за интересния случай, при който един обект може да се съживи, възкръсвайки от света на мъртвите, и да се използва отново от приложението.

Като цяло, управлението на паметта в .NET е интересна и вълнуваща тема. В настоящата глава ще се опитаме да ви дадем цялостна представа за това какво се случва в системата, докато се изпълнява управляван код, и ще навлезем в много от детайлите.

Това със сигурност ще ви помогне да разберете по-пълно .NET Framework, и може би, да пишете по-добър код.

И така, както несъмнено вече сте разбрали, управлението на паметта в .NET е автоматично. От гледна точка на разработчиците, това означава, че вече не е необходимо да се пише специален код, който да освобождава заетата от обектите памет.

---

[8] Мениджърът на памет Hoard, решава до голяма степен горните три проблема, а също има и средства за намиране на утечки на памет, но за съжаление не е безплатен.

Когато вашето приложение създава нов обект, паметта, необходима за него се заделя в регион, наречен managed heap. Заделянето на паметта и хийпът се разглеждат малко по-нататък. След като обектът е създаден, приложението използва неговата функционалност, и когато обектът стане ненужен, той просто се "изоставя", и в по-късен етап се почиства автоматично от т.нар. garbage collector – системата за почистване на паметта.

Вероятно се досещате, че работата по почистването всъщност е най-трудоемката и най-отговорна част от управлението на паметта в .NET. Алгоритъмът, по който работи garbage collector ще разгледаме подробно след малко. Засега просто приемете, че винаги, когато има недостиг от памет, се стартира системата за почистване на паметта, която идентифицира всички отпадъци – т.е. обекти, които вече не се използват от приложението и освобождава заетата от тях памет. Като програмисти по принцип нямаме контрол върху това в кой момент ще започне почистването, нито колко време ще отнеме.

Естествено, за някои обекти не е достатъчно само да се освободи паметта. Ако например даден обект капсулира файлов манипулатор, със сигурност бихме искали да освободим и този ресурс, когато вече не ни е нужен. Това не може да бъде направено автоматично от garbage collector, тъй като той се грижи само за паметта и не знае какви други системни ресурси използва обектът. За освобождаването на тези ресурси все още трябва да се погрижим ръчно. За целта в .NET съществуват т.нар. **финализатори** (finalizers) – специални методи, които се изпълняват преди обектът да се унищожи.

В горните абзаци просто нахвърляхме някои от по-важните теми, които ще бъдат разгледани повече или по-малко детайлно в главата.

Нека преди да преминем към подробностите, да се спрем на предимствата и недостатъците на тази схема на управление на паметта.

Предимства и недостатъци на автоматичното управление на паметта

Както всяка технология, така и автоматичното управление на паметта има своите плюсове и минуси. В тази секция накратко ще разгледаме по-важните от тях.

Предимства

Най-голямото предимство на автоматичното управление на паметта, разбира се е това, че ние, като разработчици, сме освободени от грижата ръчно да почистваме ненужните обекти. Това ни позволява в по-голяма степен да се съсредоточим върху бизнес логиката на нашето приложение и да отделяме по-малко време в грижи за правилното управление на паметта. Това пък, от своя страна води до по-бързо писане на кода и като цяло до съкратен цикъл на разработка.

Тясно свързано с първото е и другото голямо предимство – предотвратяването на т.нар. "memory leaks" или **изтичане на памет**. Това е много неприятен проблем, който се получава, когато разработчиците забравят да почистват ненужните обекти. В резултат, приложението започва да заема все повече памет и с течение на времето се дестабилизира. Тази ситуация е особено критична при сървърни приложения, които трябва да работят дълго време (седмици и месеци) без да се рестартират. Освен всичко друго, това е проблем, който много трудно се открива (обикновено това става, когато приложението вече се използва от клиентите) и още по-трудно се дебъгва. Понякога, при големи системи са нужни дни и дори седмици за откриването и отстраняването на причината за проблема (в много случаи причината се оказва наглед невинна грешка, и то на мястото в кода, в което сте най-сигурни че работи правилно).

В .NET можем да сме сигурни, че ако един обект не се използва от приложението, той ще бъде освободен. Сравнително трудно (но не невъзможно, както сами ще се убедите по-нататък) е да постигнете изтичане на памет.

Друг често срещан проблем, е писането и четенето по вече освободена памет или повторно освобождаване на обект. Това, в зависимост от ситуацията може да доведе до срив на цялото приложение, или до дестабилизирането му с непредвидими последици. При автоматичното управление на паметта, обектът се унищожава само когато е гарантирано недостъпен (след малко ще видим как става това), така че няма как да достъпваме обекта, ако той вече е бил унищожен.

Един от неприятните проблеми при неуправляваните приложения е липсата на съгласуваност в стратегиите за отчитане на недостиг на памет. Почти всички библиотеки използват само две стратегии, но съчетаването им ви принуждава да взимате не винаги приятни решения за дизайна на приложението ви, а също така прави кода ви труден за поддръжка. В .NET може да бъдете сигурни, че винаги ще бъде изхвърлено изключението **`OutOfMemoryException`** (въпреки, че в този момент не можете да направите кой знае какво).

При ръчното управление на паметта един от най-бележитите проблеми е този с броене на референциите към обектите, както и частния случай с циклични референции (когато два или повече обекта съдържат референции един към друг). Този проблем не съществува в .NET.

Друго, немаловажно предимство на автоматичното управление на паметта, е че паметта в хийпа се заделя много бързо. В следващата секция ще разберем защо това е така.

Е, разбира се, бързото заделяне на памет се компенсира от трудоемкото й освобождаване, което пък от своя страна е един от главните недостатъци на този модел за управление на паметта.

Недостатъци

Естествено, основният недостатък на автоматичното управление на паметта е, че почистването й е тежка и времеотнемаща операция. Когато е необходимо да се освободи памет, всички нишки на приложението заспиват и остават в това състояние докато garbage collector завърши своята работа. И тъй като системата за почистване на паметта се стартира когато има недостиг на памет, ние нямаме контрол точно в кой момент нашето приложение ще "заспи", за да се осъществи почистването, нито колко време ще трае това "заспиване".

Въпреки, че е възможно "ръчно" да контролираме работата на garbage collector чрез статичните методи на класа GC, това в огромната част от случаите е непрепоръчително, тъй като CLR обикновено може по-добре да прецени кога трябва да се осъществи почистване. Все пак, "автоматично управление" означава и по-малък контрол върху системата, което не се харесва на някои програмисти.

Алгоритъмът, по който работи garbage collector е доста добре оптимизиран и вероятно ще се оптимизира още, в бъдещите версии на .NET Framework, така че за повечето приложения, известната загуба на контрол е приемлива цена за предимствата, които получаваме. От Microsoft твърдят, че при тестове на 200 MHz Pentium машина, почистването на Поколение 0, отнема по-малко от **една милисекунда** (какво е Поколение 0 ще стане дума малко по-нататък). Така че, когато по-горе казвам че приложението ще "заспи", не оставайте с грешното впечатление, че програмите ви ще блокират за неопределен период от време – обикновено garbage collector се изпълнява достатъчно бързо за да не се забелязва с просто око.

Запомнете, че няма гаранция кога се изпълнява garbage collector и колко време отнема!

Въпреки, че е голямо предимство, високото ниво на абстракция е и огромен недостатък – неопитните програмисти, които не разбират (или по-лошо – не искат да разбират) как работи управлението на паметта в .NET и в частност системата за почистване на паметта, са способни да напишат силно неефективен по отношение на използването на паметта код, както и код, който да предизвика "изтичане" на памет дори в .NET.

Вече споменахме, че garbage collector се грижи за почистването на паметта. Все още много системни ресурси, обаче, трябва да се управляват ръчно. Не можете да очаквате от garbage collector автоматично да затвори мрежова връзка или файлов манипулатор. Когато програмирате обект, капсулиращ някакъв системен ресурс, трябва да имате това предвид и да вземете специални мерки за правилното му почистване. Как става това ще разгледаме по-нататък в настоящата тема.

Нека сега навлезем в детайлите на управлението на паметта в .NET Framework.

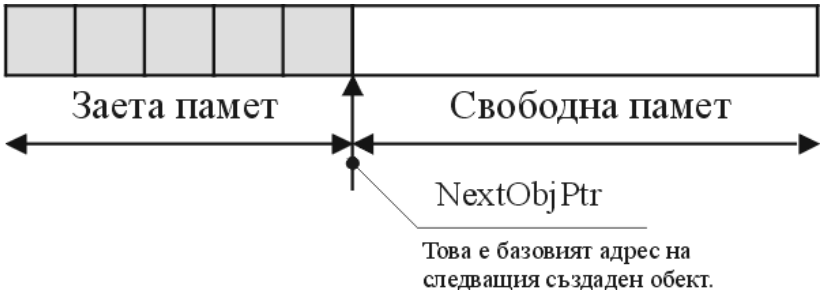
Как се заделя памет в .NET?

Когато CLR се инициализира, той заделя регион от последователни адреси в паметта. Това е т.нар. **динамична памет** или **managed heap**.

За разлика от стойностните типове, чиято памет се заделя в стека и се освобождава веднага, след като променливата излезе от обхват, паметта, нужна за референтните типове, винаги се заделя в managed heap.

В тази секция ще разгледаме как се осъществява заделянето на памет в хийпа.

В .NET, динамичната памет винаги се запълва последователно отляво надясно. Можете нагледно да си представите управлявания хийп като конвейер, при който обектите се добавят един след друг върху лентата (паметта), като всеки следващ е плътно долепен до предишния. За да е възможно това, хийпът поддържа указател, т.нар. **`NextObjPtr`**, който сочи адреса на който ще се добави следващият създаден обект. Фигурата илюстрира това описание:



Когато процесът се стартира, динамичната памет не съдържа никакви обекти и **`NextObjPtr`** е установен да сочи към базовия адрес от хийпа.

За да създадем обект в managed heap, използваме код, подобен на този:

```
SomeObject x = new SomeObject();
```

C# компилаторът превежда кода в IL **`newobj`** инструкция:

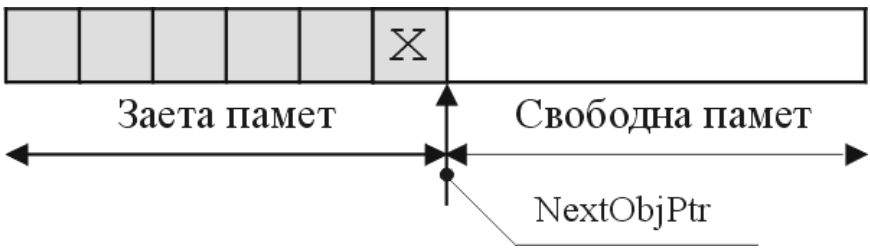
```
newobj instance void MyNamespace.SomeObject::.ctor()
```

Когато тази инструкция се изпълнява, CLR действа по следния начин:

- Изчислява размера, необходим за полетата на новия обект и всичките му родителски обекти.
- Към получения размер прибавя размера на **MethodTablePointer** и **SyncBlockIndex** (специални служебни полета). При 32-битовите системи, тези две полета добавят 8 байта към размера на всеки обект, а при 64-битовите системи – 16 байта.
- Прибавя получената стойност към указателя **NextObjPtr**. Ако в managed heap има достатъчно място, паметта се заделя, извиква се конструкторът на обекта, който я инициализира, и адресът на обекта се връща от **new** оператора. Ако CLR установи, че мястото в паметта е недостатъчно, се стартира garbage collector. След като той приключи работа, CLR опитва отново да създаде обекта. Ако и тогава няма достатъчно памет, хийпът се увеличава, а ако това е невъзможно, **new** операторът предизвиква **OutOfMemoryException**.

Значението на полетата **MethodTablePointer** и **SyncBlockIndex**, които CLR създава за всеки обект от управлявания хийп, е извън темата на тази глава. Накратко, **MethodTablePointer**, както показва името му, съдържа указател към адреса на таблицата с методите на дадения тип, а **SyncBlockIndex** се използва при синхронизацията на обекта между нишките. За целите на настоящото изложение, просто трябва да запомните, че всеки един обект от хийпа съдържа тези две полета, които увеличават размера му с 8 или 16 байта, съответно при 32 и 64 битовите системи.

След като обектът е успешно създаден, CLR установява **NextObjPtr** на първия свободен адрес, непосредствено след края на новия обект, както е показано на следващата фигура.



Вероятно се досещате, че този начин за заделяне на памет в managed heap работи много бързо, защото физически се имплементира с прибавянето на стойност (размерът на обекта) към указателя **NextObjPtr**. Всъщност скоростта на създаване на референтен тип в managed heap е съпоставима със заделянето на памет в стека. За разлика от .NET, в C++ runtime heap заделянето на памет е значително по-тежка операция, при която след изчисляването на размера на обекта първо се търси достатъчно голям блок свободна памет и едва след това обектът може да бъде създаден.

Освен това, тъй като паметта се запълва последователно, когато създаваме обекти един след друг, те физически ще се намират на близки

адреси в паметта. Това може значително да подобри производителността в някои ситуации, тъй като обектите, създадени приблизително по едно и също време обикновено са логически свързани и приложението често ги използва заедно (представете си например локални променливи в тялото на даден метод). Така е възможно всички обекти, които дадена част от кода използва, да се намират в кеша на процесора и работата с тях ще е много бърза.

Трябва да се има предвид, обаче, че освобождаването на памет от хийпа е сложна и времеотнемаща операция. Тя се извършва от системата за почистване на паметта, когато има недостиг на памет. Почистването на паметта и алгоритъмът, по който то се извършва, ще разгледаме подробно в следващите секции.

Как работи garbage collector?

В предишната секция описахме как се заделя памет, при създаването на обекти в управлявания хийп. Видяхме, че при достатъчно свободна памет това е много бърз процес, който практически се осъществява с преместването на един указател. Какво става, обаче, ако CLR установи, че в managed heap няма достатъчно място? Вече беше споменато, че ако добавянето на нов обект би довело до препълване на хийпа, трябва да се осъществи почистване на паметта. В този момент, CLR стартира системата за почистване на паметта, т.нар. garbage collector.

Всъщност това е опростено обяснение. Garbage collector се стартира когато Поколение 0 се запълни. Поколенията се разглеждат в следващата секция.

Носи се слух, че първоначално Garbage Collector в CLR е бил имплементиран на езика Lisp от Patrick Dussud, а после кода е конвертиран до C код с помощта на автоматичен конвертор и "почистен" от студент, работещ в Microsoft.

Нишките трябва да се приспят

Първото нещо, което трябва да се направи, за да може системата за почистване на паметта да започне работа, това е да се приспят всички нишки на приложението, изпълняващи управляван код. Тъй като, както след малко ще видим, по време на събирането на отпадъци е твърде вероятно обектите да се преместят на нови адреси в динамичната памет, нишките не трябва да могат да достъпват и модифицират обекти докато трае почистването.

CLR изчаква всички нишки да достигнат в безопасно състояние, след което ги приспива. Съществуват няколко механизма, чрез които CLR може да приспи дадена нишка. Причината за тези различни механизми е стремежът да се намали колкото се може повече натоварването и нишките да останат активни възможно най-дълго.

Освобождаване на неизползваните обекти

След като всички управлявани нишки на приложението са безопасно "приспани", garbage collector проверява дали в managed heap има обекти, които вече не се използват от приложението. Ако такива обекти съществуват, заетата от тях памет се освобождава. След приключване на работата по събиране на отпадъци се възобновява работата на всички нишки и приложението продължава своето изпълнение.

Както вероятно се досещате, откриването на ненужните обекти и освобождаването на ресурсите, заети от тях, не е проста задача. В тази секция накратко ще опишем алгоритъмът, който .NET garbage collector използва за нейното решаване.

За да установи кои обекти подлежат на унищожение, garbage collector построява граф на всички обекти, достъпни от нишките на приложението в дадения момент. Всички обекти от динамичната памет, които не са част от графа се считат за отпадъци и подлежат на унищожаване.

Възниква въпросът как garbage collector може да знае кои обекти са достъпни и кои не? **Корените на приложението** са точката, от която системата за почистване на паметта започва своята работа.

Корени на приложението

Всяко приложение има набор от корени (**application roots**). Корените представляват области от паметта, които сочат към обекти от managed heap, или са установени на `null`. Например всички глобални и статични променливи, съдържащи референции към обекти се считат за корени на приложението. Всички локални променливи или параметри в стека към момента, в който се изпълнява garbage collector, които сочат към обекти, също принадлежат към корените. Регистрите на процесора, съдържащи указатели към обекти, също са част от корените. Към корените на приложението спада и Freachable queue (за Freachable queue по-подробно ще стане дума в секцията за финализация на обекти в настоящата глава. Засега просто приемете че тази опашка е част от вътрешните структури, поддържани от CLR и се счита за един от корените на приложението).

Когато JIT компилаторът компилира IL инструкциите на даден метод в процесорни инструкции, той също съставя и вътрешна таблица, съдържаща корените за съответния метод. Тази таблица е достъпна за garbage collector. Ако се случи garbage collector да започне работа, когато методът се изпълнява, той ще използва тази таблица, за да определи кои са корените на приложението към този момент. Освен това се обхожда и стекът на извикванията за съответната нишка и се определят корените за всички извикващи методи (като се използват техните вътрешни таблици). Към получения набор от корени, естествено, се включват и тези, намиращи се в глобални и статични променливи.

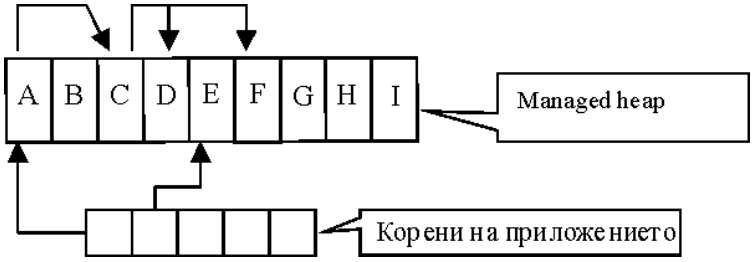
Трябва да се помни, че не е задължително даден обект да излезе от обхват за да бъде считан за отпадък. JIT компилаторът може да определи

кога този обект се достъпва от кода за последен път и веднага след това го изключва от вътрешната таблица на корените, с което той става кандидат за почистване от garbage collector. Изключение правят случаите, когато кодът е компилиран с **/debug** опция, която предотвратява почистването на обекти, които са в обхват. Това се прави за улеснение на процеса на дебъгване – все пак при трасиране на кода бихме искали да можем да следим състоянието на всички обекти, които са в обхват в дадения момент.

Алгоритъмът за почистване на паметта

Когато garbage collector започва своята работа, той предполага че всички обекти в managed heap са отпадъци, т.е. че никой от корените не сочи към обект от паметта. След това, системата за почистване на паметта започва да обхожда корените на приложението и да строи граф на обектите, достъпни от тях.

Нека разгледаме примера, показан на следващата фигура. Ако глобална променлива сочи към обект A от managed heap, то A ще се добави към графа. Ако A съдържа указател към C, а той от своя страна към обектите D и F, всички те също стават част от графа. Така garbage collector обхожда рекурсивно в дълбочина всички обекти, достъпни от глобалната променлива A:



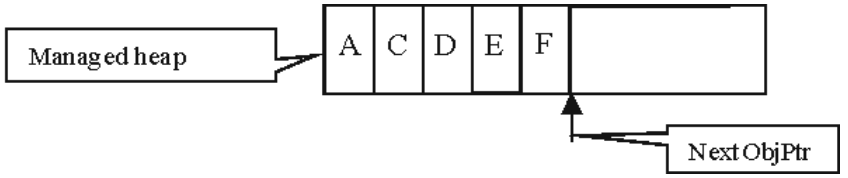
Когато приключи с построяването на този клон от графа, garbage collector преминава към следващия корен и обхожда всички достъпни от него обекти. В нашия случай към графа ще бъде добавен обект E. Ако по време на работата garbage collector се опита да добави към графа обект, който вече е бил добавен, той спира обхождането на тази част от клона. Това се прави с две цели:

- значително се увеличава производителността, тъй като не се преминава през даден набор от обекти повече от веднъж;
- предотвратява се попадането в безкраен цикъл, ако съществуват циклично свързани обекти (например A сочи към B, B към C, C към D и D обратно към A).

След обхождането на всички корени на приложението, Графът съдържа всички обекти, които по някакъв начин са достъпни от приложението. В посочения на фигурата пример, това са обектите A, C, D, E и F.

Всички обекти, които не са част от този граф, не са достъпни и следователно се считат за отпадъци. В нашия пример това са обектите B, G, H и I.

След идентифицирането на достъпните от приложението обекти, garbage collector преминава през хийпа, търсейки последователни блокове от отпадъци, които вече се смятат за свободно пространство. Когато такава област се намери, всички обекти, намиращи се над нея се придвижват надолу в паметта, като се използва стандартната функция **`memcpy(…)`**. Крайният резултат е, че всички обекти, оцелели при преминаването на garbage collector, се разполагат в долната част на хийпа, а **`NextObjPtr`** се установява непосредствено след последния обект. Фигурата показва състоянието на динамичната памет след приключване на работата на garbage collector:



Описаният алгоритъм за почистване на паметта не взима предвид финализацията. Обектите, нуждаещи се от финализация не се унищожават веднага. Вместо това те остават в паметта и указатели към тях се добавят във т. нар. Freachable queue. Финализацията ще разгледаме подробно малко по-нататък.

Естествено, преместването на обект на друго място в паметта прави невалидни всички указатели, сочещи към него, така че част от "задълженията" на garbage collector е да коригира по подходящ начин указателите към оцелелите обекти.

Пренареждането на хийпа е трудоемка операция – трябва да се придвижват големи области от паметта и да се валидират указателите към преместените обекти. Затова ако garbage collector срещне малка област от незаета памет, той просто я игнорира и продължава нататък.

Като цяло, работата на garbage collector има значително отражение върху производителността на цялото приложение. Построяването на графа на достъпните обекти, обхождането и пренареждането на динамичната памет отнемат немалко процесорно време, през което нишките на приложението спят. Трябва да се има предвид, обаче, че garbage collector се стартира само когато има нужда от това (т.е. когато има недостиг на памет). През останалото време managed heap е доста по-бърз от C/C++ runtime heap.

В помощ на производителността са и някои оптимизации на алгоритъма на garbage collector, най-важната от които е концепцията за **поколения**. Нека разгледаме поколенията памет.

Поколения памет

Поколенията (generations) са механизъм в garbage collector, чиято единствена цел е подобряването на производителността. Основната идея е, че почистването на част от динамичната памет винаги е по-бързо от почистването на цялата памет. Вместо да обхожда всички обекти от хийпа, garbage collector обхожда само част от тях, класифицирайки ги по определен признак. В основата на механизма на поколенията стоят следните предположения:

- колкото по-нов е един обект, толкова по-вероятно е животът му да е кратък. Типичен пример за такъв случай са локалните променливи, които се създават в тялото на даден метод и излизат от обхват при неговото напускане.
- колкото по-стар е обектът, толкова по-големи са очакванията той да живее дълго. Пример за такива обекти са глобалните променливи.
- обектите, създадени по едно и също време обикновено имат връзка помежду си и имат приблизително еднаква продължителност на живота.

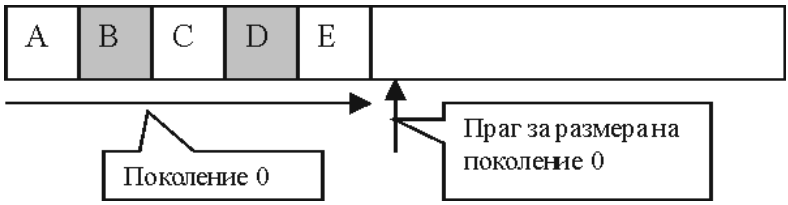
Много изследвания потвърждават валидността на изброените твърдения за голям брой съществуващи приложения. Нека разгледаме по-подробно поколенията памет и това как те се използват за оптимизация на производителността на .NET garbage collector.

Поколение 0

Когато приложението се стартира, първоначално динамичната памет не съдържа никакви обекти. Всички обекти, които се създават, стават част от Поколение 0. Казано накратко Поколение 0 съдържа новосъздадените обекти – тези, които никога не са били проверявани от garbage collector.

При инициализацията на CLR се определя праг за размера на Поколение 0. Точният размер на този праг не е от особено значение, тъй като може да се променя от garbage collector по време на работа с цел подобряване на производителността. Да предположим, че първоначално стойността на този праг е 256KB.

Следващата фигура показва състоянието на динамичната памет след като приложението е работило известно време. Виждаме, че са създадени известен брой обекти (всички част от Поколение 0), а обекти B и D вече са станали недостъпни (т.е. подлежат на почистване).



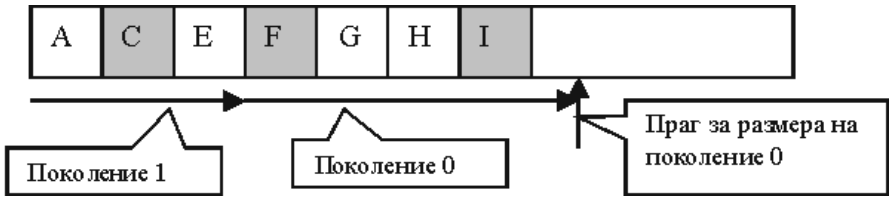
Да предположим, че приложението иска да създаде нов обект, F. Добавянето на този обект би предизвикало препълване на Поколение 0. В този момент трябва да започне събиране на отпадъци и се стартира garbage collector.

Почистване на Поколение 0

Garbage collector процедира по описания по-горе алгоритъм и установява че обекти B и D са отпадъци. Тези обекти се унищожават и оцелелите обекти A, C и E се пренареждат в долната (или лява) част на managed heap. Динамичната памет непосредствено след приключването на събирането на отпадъци изглежда по следния начин:

Сега оцелелите при преминаването на garbage collector обекти стават част от Поколение 1 (защото са оцелели при едно преминаване на garbage collector). Новият обект F, както и всички други новосъздадени обекти ще бъдат част от Поколение 0.

Нека сега предположим, че е минало още известно време, през което приложението е създавало обекти в динамичната памет. Managed heap сега изглежда по следния начин:

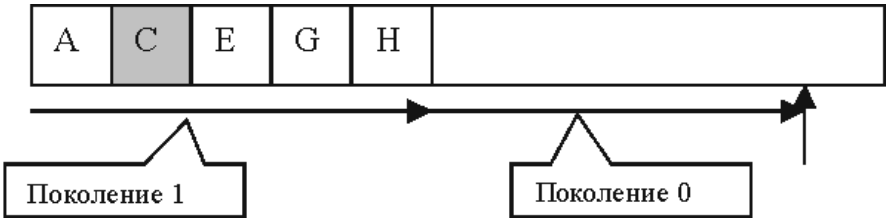


Добавянето на нов обект J, би предизвикало препълване на Поколение 0, така че отново трябва да се стартира събирането на отпадъци. Когато garbage collector се стартира, той трябва да реши кои обекти от паметта да прегледа. Както Поколение 0, така и Поколение 1 има праг за своя размер, който се определя от CLR при инициализацията. Този праг е по-голям от този на Поколение 0. Да предположим че той е 2MB.

В случая Поколение 1 не е достигнало прага си, така че garbage collector ще прегледа отново само обектите от Поколение 0. Това се диктува от правилото, че по-старите обекти обикновено имат по-дълъг живот и следователно почистването на Поколение 1 не е вероятно да освободи много памет, докато в Поколение 0 е твърде възможно много от обектите да са отпадъци. И така, garbage collector почиства отново Поколение 0, оцелелите обекти преминават в Поколение 1, а тези, които преди това са били в Поколение 1, просто си остават там.

Забележете, че обект C, който междувременно е станал недостъпен и следователно подлежи на унищожение, в този случай остава в динамичната памет, тъй като е част от Поколение 1 и не е проверен при това преминаване на garbage collector.

Следващата фигура показва състоянието на динамичната памет след това почистване на Поколение 0.



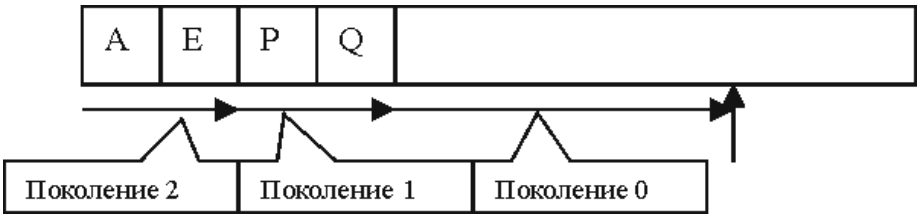
Както вероятно се досещате, с течение на времето Поколение 1 бавно ще расте. Идва момент, когато след поредното почистване на Поколение 0, Поколение 1 достига своя праг от 2 MB. В този случай приложението просто ще продължи да работи, тъй като Поколение 0 току-що е било почистено и е празно. Новите обекти, както винаги, ще се добавят в Поколение 0.

Почистване на Поколение 1 и Поколение 2

Когато Поколение 0 следващият път достигне своя праг и garbage collector се стартира, той ще провери размера на Поколение 1. Тъй като той е достигнал своя праг от 2 MB, garbage collector този път ще почисти както Поколение 0, така и Поколение 1. Забележете, че след като са минали няколко почиствания на Поколение 0, с течение на времето, е твърде вероятно Поколение 1 да съдържа много обекти, които са станали недостъпни и неговото почистване би освободило голямо количество памет.

И така, garbage collector почиства поколения 0 и 1. Обектите, оцелели от Поколение 0 преминават в Поколение 1, а тези, които преди това са били в Поколение 1 и са оцелели при почистването преминават в Поколение 2.

Следващата фигура показва примерното състояние на динамичната памет след почистването на поколения 0 и 1 (предполагаме, че обекти G и H с течение на времето са станали недостъпни и са били почистени от garbage collector, а обекти P и Q са нови обекти, оцелели от Поколение 0 и преминали в Поколение 1).



Текущата версия на CLR garbage collector поддържа три поколения – 0, 1 и 2. Обектите, които оцелеят при почистване на Поколение 2 просто си остават в Поколение 2.

Разбира се, Поколение 2 също има праг за своя размер и той е около 10 MB.

Поколение 0 се почиства най-често – в него се съдържат нови обекти и е най-вероятно те да имат кратък живот.

Поколение 2 се почиства най-рядко. Това поколение съдържа само стари обекти, преживели 2 или повече проверки от garbage collector.

Имплементация на поколенията в .NET

Както видяхме, поколенията значително подобряват производителността на garbage collector. Ако докато строи графа на достъпните обекти, garbage collector срещне референция към обект от по-горно поколение, той просто не продължава да строи тази част от клона. Това е безопасно, защото при преминаването през хийпа, garbage collector преглежда само обектите от поколението, което се почиства, следователно няма опасност да се унищожат обекти от горните поколения, дори и да не са част от графа.

Какво ще се случи, обаче, ако обект от по-старо поколение държи референция към по-млад обект? Ако референциите на стария обект не се проследят, младият обект погрешно ще бъде счетен за недостъпен, няма да бъде добавен към графа и ще бъде унищожен!

За да се избегнат подобни проблеми, JIT компилаторът поддържа механизъм, който установява флаг, когато някое от референтните полета на даден обект се промени. Така garbage collector може да установи референциите на кои обекти са променени от времето на последното събиране на отпадъци. Тези стари обекти ще бъдат инспектирани от garbage collector, за да се провери дали не съдържат референции към по-млади обекти.

Вече споменахме, че garbage collector динамично може да променя праговете за размера на отделните поколения. Ако например с течение на времето, системата установи, че при почистването на Поколение 0 оцеляват много малко обекти, прагът на Поколение 0 може да се намали, да речем на 128 KB. Така почистванията на Поколение 0 ще са по-чести, но ще отнемат по-малко време. При обратния случай – ако почистването на Поколение 0 освобождава много малко памет, а оцелелите са много, прагът ще бъде увеличен например на 512 KB. Така събирането на отпадъци ще е по-рядко и ще има по-голяма вероятност междувременно много обекти да станат недостъпни.

Горното важи, разбира се и за праговете на Поколения 1 и 2. Те също подлежат на промяна с цел оптимизация от страна на garbage collector.

Workstation и Server GC

В CLR всъщност съществуват две разновидности на garbage collector – **Server GC** и **Workstation GC**. Във версии 1.0 и 1.1 на .NET Framework, тези две разновидности се съдържат в двете библиотеки **`MSCorSvr.dll`** (Server GC) и **`MSCorWks.dll`** (Workstation GC). В Whidbey - версия 2.0 на .NET Framework, двете библиотеки са обединени в една.

Конзолните и Windows приложенията използват Workstation GC, който е оптимизиран за минимизиране на времето, през което нишките на приложението са приспани. Тъй като потребителят не трябва да вижда забележима пауза в работата на приложението, garbage collector построява графа на достъпните обекти докато нишките на приложението още работят. Нишките се приспиват едва, когато garbage collector започне истинското почистване на managed heap. Това е т.нар. конкурентно почистване на паметта.

Server GC се използва за сървърни приложения при многопроцесорни машини. В този случай, за всеки отделен процесор се построява отделен хийп, за чието почистване се грижи отделна нишка на garbage collector. Хийповете на отделните процесори се почистват паралелно, като през цялото време нишките на приложението спят. Тази техника показва добра производителност при многопроцесорни машини и има много по-добра скалируемост.

По подразбиране, режимът на работа на garbage collector е Workstation. При еднопроцесорните машини, това е единствения избор. В .NET Framework 1.1 SP1 и 2.0 съществува възможността режимът на работа на garbage collector да се посочи в конфигурационния файл на приложението по следния начин:

```
<Configuration>
    <runtime>
        <gcServer enabled="true" />
    </runtime>
</Configuration>
```

Блок памет за големи обекти

Размерът е без значение. ***Йода***

Друга важна оптимизация, свързана с .NET Framework managed heap е т. нар. блок памет за големи обекти (large object heap, LOH). С цел подобряване на производителността всички големи обекти (с размер над 20 000 байта) се разполагат в отделен хийп. Разликата между него и стандартния managed heap е това, че хийпът за големи обекти не се дефрагментира. Преместването на тези големи блокове от паметта просто би отнело прекалено много процесорно време.

Всичко това става прозрачно за разработчиците. От гледна точка на приложението, нещата изглеждат така, сякаш има един единствен хийп.

Имайте предвид, че големите обекти винаги се считат за част от Поколение 2. Това означава, че по-възможност трябва да създаваме по-малко на брой големи обекти и да ги използваме в случаите, когато те ще живеят дълго време.

Създаването на голям брой големи обекти с кратък живот ще доведе до това, че Поколение 2 по-често ще достига прага за своя размер и по-често ще се почиства, което пък значително ще влоши производителността.

Увеличаване размера на хийпа

В случай, че след почистване на всички поколения, все още няма достатъчно памет за създаване на даден обект, необходим на приложението, CLR ще увеличава размера на managed heap и съответният процес, в който се изпълнява CLR, започва да заема повече памет от операционната система. Ако е необходимо, се използва виртуалната памет.

Виртуалната памет се съхранява на твърдия диск. Когато операционната система има нужда от памет, а физическата RAM памет на компютъра не е достатъчна, се извършва процес, при който неактивни страници от RAM паметта, се прехвърлят на твърдия диск. Когато тези страници от паметта трябва да се достъпят отново, те се копират обратно в RAM. Естествено дискът е много по-бавен от истинската RAM памет, така, че целият този процес може да отнеме доста време, през което приложенията работят много бавно (дори за известен период могат да спрат да опресняват интерфейса си и да изглеждат "увиснали").

Финализацията на обекти в .NET

Както видяхме, garbage collector ни освобождава напълно от грижите по управлението и почистването на паметта. В света на .NET, вече са невъзможни изтичането на памет и обръщението към вече унищожен обект, две особено неприятни грешки, които много трудно се откриват и проследяват и могат да превърнат поддръжката или дебъгването на една система в кошмар за програмистите.

За радост, повечето от обектите, с които нашите приложения ще работят, изискват само памет за да са функционални. Например **`Int32`**, **`Double`**, **`String`** и **`Hashtable`** са типове, които съхраняват и манипулират байтове от паметта. За тези и за много други обекти, спокойно можем да оставим да се погрижи garbage collector. Не е необходимо да извършваме каквито и да било действия по почистването на ресурсите, заети от тях. Вместо това, ние просто създаваме обекта, използваме неговата функционалност и когато вече не ни трябва, можем да бъдем сигурни, че в по-късен етап garbage collector ще се погрижи да освободи заетата от него памет.

При други обекти, обаче, нещата са малко по-сложни. Например типът **`System.IO.FileStream`** вътрешно съдържа файлов манипулатор, който се

използва от методите му **`Read(…)`** и **`Write(…)`**. По подобен начин, **`System.Data.OleDb.OleDbConnection`** капсулира връзка към база от данни, а **`System.Net.Sockets.Socket`** – мрежов сокет.

За всички подобни обекти, капсулиращи някакъв ценен системен ресурс, трябва да се вземат специални мерки, тъй като сам по себе си, garbage collector не може да освобождава тези ресурси. Това е отговорност на самия обект. Именно в тази ситуация на помощ идва **финализацията**.

Какво е финализация?

Накратко, финализацията позволява да се почистват ресурси, свързани с даден обект, преди обектът да бъде унищожен от garbage collector. Обяснено най-просто, това е начин да се каже на CLR "преди този обект да бъде унищожен, трябва да се изпълни ето този код".

За да е възможно това, класът трябва да имплементира специален метод, наречен **`Finalize()`**. Когато garbage collector установи, че даден обект вече не се използва от приложението, той проверява дали обектът дефинира **`Finalize()`** метод. Ако това е така, **`Finalize()`** се изпълнява и на по-късен етап (най-рано при следващото преминаване на garbage collector), обектът се унищожава. Този процес ще бъде разгледан детайлно след малко. Засега просто трябва да запомните две неща:

- **`Finalize()`** **не може да се извиква явно**. Този метод се извиква само от системата за почистване на паметта, когато тя прецени, че даденият обект е отпадък.
- Най-малко **две** преминавания на garbage collector са необходими за да се унищожи обект, дефиниращ **`Finalize()`** метод. При първото се установява че обектът подлежи на унищожение и се изпълнява финализаторът, а при второто се освобождава и заетата от обекта памет. Всъщност в реалния живот почти винаги са необходими повече от две събирания на garbage collector поради преминаването на обекта в по-горно поколение.

Деструкторите в C#

В .NET, класът **`System.Object`** дефинира **`Finalize()`** метод. Ако искаме да осигурим финализатор за нашия клас, бихме използвали следния код:

```
protected override void Finalize()
{
   try
   {
      // Cleanup code goes here
   }
   finally
   {
      base.Finalize();
```

```
    }
}
```

Както виждате, това, което правим, е да предефинираме **Finalize()** метода на класа **System.Object** (спомнете си, че всички типове в .NET наследяват **System.Object**). Използваме конструкцията **try … finally** за да се подсигурим, че независимо какъв е резултатът от изпълнението на почистващия код, ще бъде извикан **Finalize()** методът на родителския обект.

Забележете, че макар System.Object да дефинира Finalize() метод, за да поддържа финализация, вашият клас, или някой от родителските му типове трябва да припокрива Finalize() метода (чрез използването на деструктор). Т.е. ако Finalize() методът на вашия клас е този, наследен от System.Object, то инстанциите на класа няма да поддържат финализация.

Всъщност, ако се опитате да компилирате показания по-горе код, ще получите следното съобщение за грешка от C# компилатора:

```
Do not override object.Finalize. Instead, provide a destructor.
```

Дефиниране на деструктори в C#

Екипът, разработвал C# компилатора, установява, че много програмисти не имплементират **Finalize()** правилно. По-специално, мнозина забравят да използват **try … finally** блок и да извикат **base.Finalize()**. Поради тази причина, в C# не може **Finalize()** да се имплементира явно. Вместо това се използват **деструктори**, които имат следния специален синтаксис:

```
~MyClass ()
{
   // Cleanup code goes here
}
```

Този код се преобразува от компилатора във **Finalize()** метод, по такъв начин, че става напълно еквивалентен на предишния (т.е. автоматично се добавя **try…finally** и се извиква **base.Finalize()** във **finally** блока).

Забележете, че макар документацията на C# да използва терминът деструктор, а синтаксисът да е еквивалентен на деструкторите в C++, всъщност приликата свършва до тук. В C# деструкторите се преобразуват във Finalize() методи, които се извикват от системата за почистване на па-

	метта. Унищожаването на обектите не е детерминистично и програмистът няма възможност да определи кога и в какъв ред се изпълняват финализаторите. При някои специални обстоятелства дори няма гаранция, че те изобщо ще се изпълнят. Запомнете: общото между деструкторите в C# и тези в C++ се изчерпва със синтаксиса.

Финализация – пример

Нека обобщим казаното досега в един по-завършен пример. В кода показан по-долу, дефинираме клас, който капсулира някакъв Windows ресурс (манипулатор към който се съхранява в член-променливата **`mResourceHandle`**):

```
using System;

// Wrapper around Windows resource
class ResourceWrapper
{
   private IntPtr mResourceHandle = IntPtr.Zero;

   public ResourceWrapper()
   {
      // Allocate the resource here
   }

   ~ResourceWrapper()
   {
      if (mResourceHandle != IntPtr.Zero)
      {
         // Deallocate the resource here
         // ...
         mResourceHandle = IntPtr.Zero;
      }
   }
}
```

Забележете, че кодът, показан тук, е просто пример как трябва да се дефинира деструктор, но не е правилният начин за освобождаване на системни ресурси. По причини, които ще изясним след малко, не е ефективно да се разчита само на финализацията, когато трябва да се освободи системен ресурс. По-нататък, в секцията "Ръчно управление на ресурсите с интерфейса `IDisposable`" ще дадем пример как точно трябва да се подходи в такъв случай.

Зад кулисите

Нека сега разгледаме малко по-подробно какво всъщност се случва, когато дефинираме деструктор в кода на нашия клас. В тази секция ще

изложим кратко описание на процесите, които протичат зад кулисите, когато CLR изпълнява кода. След това ще дадем някои препоръки, свързани с използването на `Finalize()` методи.

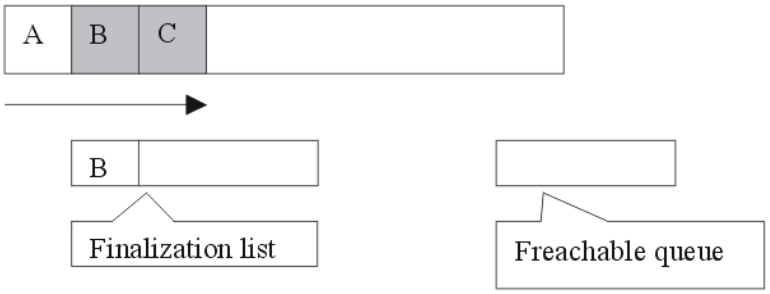
И така, CLR поддържа две структури, които са свързани с финализацията. Това са т.нар. **Finalization List** и **Freachable Queue**.

Когато се създава нов обект, CLR проверява дали типът дефинира `Finalize()` метод и ако това е така, след създаването на обекта в динамичната памет (но преди извикването на неговия конструктор), указател към обекта се добавя към Finalization list. Така Finalization list съдържа указатели към всички обекти в хийпа, които трябва да бъдат финализирани (имат `Finalize()` методи), но все още се използват от приложението (или вече не се използват, но още не са проверени от garbage collector).

Създаването на обект, поддържащ финализация изисква една допълнителна операция от страна на CLR – поставянето на указател във Finalization list и следователно отнема и малко повече време.

Взаимодействието на garbage collector с обектите, нуждаещи се от финализация, е твърде интересно. Нека разгледаме следния пример.

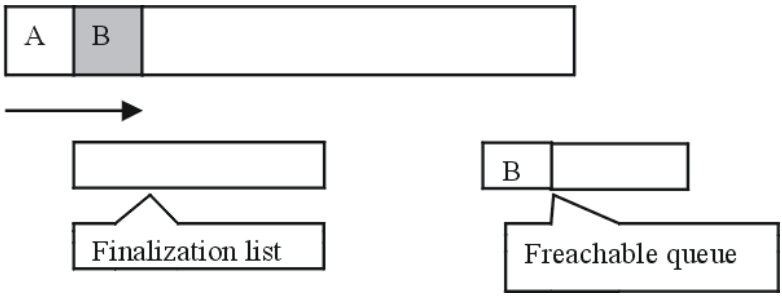
Фигурата по-долу показва опростена схема на състоянието на динамичната памет точно преди да започне почистване на паметта. Виждаме че хийпът съдържа три обекта – A, B и C. Нека всички те са от Поколение 0. Обект A все още се използва от приложението, така че той ще оцелее при преминаването на garbage collector. Обекти B и C, обаче, са недостъпни от корените и се определят от garbage collector-а като отпадъци.



Когато даден обект се идентифицира като отпадък, garbage collector проверява дали във Finalization list съществува указател към този обект. Когато такъв указател няма (какъвто е случаят с обект C), неговата памет просто може да се освободи по начина, вече описан в секцията "Как работи garbage collector?".

Когато обаче във Finalization list се намери такъв указател (както в случая с обект B), garbage collector не може просто да унищожи обекта, тъй като преди това трябва да се извика неговия `Finalize()` метод. Вместо това, указателят към обекта ще бъде изтрит от Finalization list и ще бъде добавен към **Freachable queue**.

Състоянието на динамичната памет непосредствено след приключването на събирането на отпадъци е следното:



На фигурата впечатление правят две неща:

- Обект C е унищожен и паметта, заемана от него, може да се използва повторно от приложението.
- Указателят към обект B е преместен от Finalization list във Freachable queue, а самият обект продължава да "живее" в динамичната памет и тъй като е оцелял при преминаването на garbage collector, вече е **част от Поколение 1**.

Опашката Freachable

Опашката Freachable съдържа указатели към всички обекти, чиито `Finalize()` методи вече могат да се извикат. Името на тази опашка всъщност означава следното: F е съкратено от Finalization – всеки елемент от опашката е указател към обект, който трябва да се финализира, а reachable (достъпен) означава, че обектът е *достъпен от приложението*. Всеки обект, за който има запис във Freachable queue **е достъпен от приложението и не е отпадък**. Т.е. Freachable queue се счита за **част от корените** на приложението, както например са глобалните и статични променливи.

Накратко за финализацията

И така, garbage collector първо определя обект B като недостъпен и следователно – подлежащ на почистване. След това указателят към обект B се изтрива от Finalization list и се добавя към опашката Freachable. В този момент обектът се **съживява**, т.е. той се добавя към графа на достъпните обекти и вече не се счита за отпадък. Garbage collector пренарежда динамичната памет. При това обект B се третира както всеки друг достъпен от приложението обект, в нашия пример – обект A.

След това CLR стартира специална нишка с висок приоритет, която за всеки запис във Freachable queue изпълнява `Finalize()` метода на съответния обект и след това **изтрива записа от опашката**.

При следващото почистване на Поколение 1 от garbage collector, обект B ще бъде третиран като недостъпен (защото записът вече е изтрит от

Freachable queue и никой от корените на приложението не сочи към обекта) и паметта, заемана от него ще бъде освободена. Забележете, че тъй като обектът вече е в по-високо поколение, преди това да се случи е възможно да минат още няколко преминавания на garbage collector.

Тъмната страна на финализацията

Люк: Не съм уплашен.

Йода: Добре. Ще бъдеш. Ще бъдеш.

Финализацията е неефективна

Обектите, поддържащи финализация, престояват в паметта значително по-дълго от останалите обекти. Освен това, представете си, че обектът B от горния пример държи референции към други обекти. Тези обекти (и обектите, реферирани пряко или непряко от тях) също ще се съживят и ще останат в паметта, докато живее и обект B. Това означава, че garbage collector не може да освободи паметта в момента, когато установи че обектите са отпадъци.

Неправилно е да се разчита (само) на финализацията, за освобождаване на системни ресурси. Помнете, че нямате контрол върху това кога ще се изпълнят финализаторите. Това може да доведе до ненужно дълго задържане на ресурсите заети.

Проблеми с нишките

`Finalize()` методите се изпълняват от отделна нишка на CLR. Следователно, във финализаторите не трябва да се пише код, който прави каквито и да било предположения относно нишката в която се изпълнява.

Освен това, трябва да се избягва код, отнемащ много време. В най-лошия случай, ако даден **`Finalize()`** метод влезе в безкраен цикъл, нишката, изпълняваща финализаторите ще се блокира и останалите **`Finalize()`** методи няма да се изпълнят. Това е много опасна ситуация, защото докато приложението работи, garbage collector няма да може да освободи заеманата от тези обекти памет.

При прекратяване на работата на приложението, когато CLR се изключва, на всеки **`Finalize()`** метод се дават приблизително две секунди, за да се изпълни. Ако методът не завърши изпълнението си за това време, CLR просто убива процеса и не изпълнява повече финализатори. Освен това, CLR дава приблизително четиридесет секунди за да се изпълнят всички **`Finalize()`** методи. След като това време мине, процесът се убива.

Проблеми с реда на изпълнение на финализаторите

CLR не дава никакви гаранции за реда, в който ще се извикат отделните финализатори. Това означава, че не е безопасно във **`Finalize()`** метод да се обръщате към друг обект, поддържащ финализация, защото неговият финализатор може вече да е бил изпълнен и състоянието на обекта в този

случай е непредвидимо. Също, не е безопасно да извиквате статични методи. Тези методи вътрешно могат да използват обекти, които вече са били финализирани и резултатите отново са непредсказуеми.

Какво да правим все пак?

След всичко казано дотук, може би вече се чудите, за какво можете да използвате **`Finalize()`** методите. И след като има толкова много неща които **не трябва** или не е ефективно да се правят във финализацията, какво да правим когато се налага да освободим някакъв ресурс?

Microsoft препоръчва използването на финализацията да става съвместно с имплементирането на интерфейса **`IDisposable`**.

Не разчитайте само на финализацията за да освобождавате ресурси. Имплементирайте `IDisposable` и използвайте `Finalize()` методите съвместно с него.

Съживяване на обекти

Смъртта е естествена част от живота. Радвайте се за онези които се превръщат в част от Силата. Не ги оплаквайте. Нека не ви липсват. Привързването води до ревност. То е сянка на алчността. ***Йода***

Както видяхме, цикълът на живот на обект, нуждаещ се от финализация е интересен. Обектът умира, след това референция към него се добавя към един от корените на приложението (Freachable queue) при което обектът се съживява, неговият **`Finalize()`** метод се изпълнява, указателят се изтрива от Freachable queue и обектът умира завинаги. В по-късен етап, garbage collector просто ще освободи заетата от този "мъртъв" обект памет.

Какво ще се случи, обаче, ако по време на финализацията обектът се добави към някой от корените на приложението?

Разгледайте следния код:

```
public class ClassThatResurrects
{
   ~ClassThatResurrects()
   {
      SomeRootClass.mThisIsARoot = this;
   }
}

public class SomeRootClass
{
   public static object mThisIsARoot;
```

```
}
```

В този пример по време на финализацията, обект от типа **ClassThatResurrects** ще добави референция към себе си в един от корените на приложението – статичната член-променлива **mThisIsARoot**. Така обектът става достъпен и garbage collector няма да го унищожи. Приложението може да използва обекта чрез променливата **mThisIsARoot** докато тя сочи към него. Това е интересен случай на съживяване на обект. Виждаме, че като .NET програмисти имаме почти "свръхестествената" сила да връщаме и използваме обекти от света на мъртвите.

Когато в някакъв момент на **mTisIsARoot** се присвои указател към друг обект, или просто **null**, съживеният обект отново умира и ще бъде почистен (някога) от garbage collector. В този случай обектът няма повече да се финализира, тъй като неговият **Finalize()** метод вече е бил извикан веднъж и указател към обекта вече не съществува във Finalization list.

Ако все пак искаме **Finalize()** методът да се изпълни отново, garbage collector предлага статичният метод **ReRegisterForFinalize()**, който приема един единствен параметър – референция към обект. Извикването на този метод добавя указател към обекта във Finalization list. Когато garbage collector прецени, че обектът е отпадък, указателят ще бъде преместен във Freachable queue и **Finalize()** метода ще бъде извикан отново. Ето и пример:

```
~ClassThatResurrects()
{
   SomeRootClass.thisIsARoot = this;
   GC.ReRegisterForFinalize(this);
}
```

Всъщност при написан по този начин деструктор, обектът ще се съживява всеки път когато се извика неговият **Finalize()** метод и докато приложението работи, той никога няма да умре. Когато приложението прекратява работата си, CLR ще изчака определено време и след това просто ще убие процеса. В едно реално приложение вероятно бихте правили някаква проверка, преди да пререгистрирате обекта за финализация.

Също, **никога не извиквайте `ReRegisterForFinalize()` повече от веднъж в деструктора**, това ще доведе до многократното му изпълнение при следващото почистване.

Например този код ще доведе до това че деструкторът ще се изпълни два пъти при следващото съживяване на обекта (четири пъти при следващото, осем след това и т.н.). **Никога не правете това в реална програма:**

```
~ClassThatResurrects()
{
   SomeRootClass.thisIsARoot = this;
```

```
    GC.ReRegisterForFinalize(this);
    GC.ReRegisterForFinalize(this);
}
```

Причината за това поведение на кода е, че **`ReRegisterForFinalize()`** просто добавя указател към обекта във Finalization list без да проверява дали такъв вече съществува.

Трябва да се помни, че използването на съживяване **не е препоръчителна практика и трябва да се избягва винаги, когато има възможност**. Всъщност макар да звучи интересно и вълнуващо, случаите, в които прилагането на съживяване е добра идея, са доста малко. Използването на вече финализирани обекти може да има непредсказуеми резултати. Ако например съживеният обект вътрешно се обръща към други обекти, изискващи финализация, **`Finalize()`** методите на някои от тях могат вече да са били изпълнени. Освен това, всички обекти, към които даденият обект има референции, също ще се съживят и ще заемат памет от хийпа.

Ръчно управление на ресурсите с IDisposable

Люк: Бъдещето ли? Ще умрат ли?

Йода: Трудно е да се види. Бъдещето винаги е в движение.

Голям недостатък на системата за почистване на паметта е, че няма абсолютно никаква гаранция кога ще бъде почистена паметта, нито кога ще бъдат извикани финализаторите на недостижимите обекти.

Обикновено това не бива да ви притеснява, но ако вашия обект е обвивка около неуправляван ресурс на операционната система (например файл, сокет, връзка към база данни и т.н.) в повечето случаи ще искате да освободите ресурса, за да може да го използват и други приложения или друга нишка във вашето приложение.

Не само това, но и ръчното управление на ресурсите може да бъде по-ефективно. Например, ако пишете код използващ графичен ресурс (например четка или шрифт) и знаете, че от един определен момент ресурсът вече не ви е необходим, не само е добре да го освободите, тъй като той консумира неявно памет, но и е почти задължително да го освободите, тъй като почти всички ресурси на операционната система са ограничени[9].

Както може да очаквате, в .NET Framework съществува решение на проблема.

[9] Това естествено зависи от наличието на памет, тъй като не ресурсите, а манипулаторите (handles) за ресурсите са ограничени

Интерфейсът IDisposable

Интерфейсът **IDisposable** се препоръчва от Microsoft в тези случаи, в които искате да гарантирате моментално освобождаване на ресурсите (вече знаете, че използването на **Finalize()** не го гарантира).

Използването на **IDisposable** се състои в имплементирането на интерфейса от класа, който обвива някакъв неуправляван ресурс и освобождаването на ресурса при извикване на метода **Dispose()**. Ето как изглежда този интерфейс:

```
public interface IDisposable
{
   void Dispose();
}
```

Използването на **IDisposable** обекти е тривиално, но ще ви покажем правилната употреба, когато очаквате възникване на изключение, тъй като сме виждали няколко погрешни практики, които не искаме да научите:

```
// Придобиваме ресурса
Resource resource = new Resource();
try
{
   // Използваме ресурса
}
finally
{
   // Унищожаваме (освобождаваме) ресурса.
   // Преобразуването на обекта към IDisposable е добра практика
   // тъй като обектът може да имплементира интерфейса
   // експлицитно както ще видите по-нататък
   ((IDisposable)resource).Dispose();
}
```

Операторът using

Тази употреба се среща толкова често, че Microsoft са добавили в езика C# оператора **using**, който прави същото без изричното споменаване на оператори и клаузи за работа с изключения:

```
Resource resource = new Resource();
using (resource)
{
   // Използваме ресурса
}
```

Операторът се превръща от C# компилатора в следния код:

```
Resource resource = new Resource();
try
{
}
finally
{
   if (resource != null)
   {
       ((IDisposable)resource).Dispose();
   }
}
```

Естествено, компилаторът е достатъчно интелигентен, за да премахне проверката за **null** преди да извика **Dispose()** в простите случаи като горния.

Ние ви препоръчваме да създадете ресурса в израза, поместен в скобите на оператора **using**, като по този начин ще намалите видимостта на променливата **resource** (което и ще искате да направите в повечето от случаите):

```
// Придобиваме ресурса
using (Resource resource = AcquireResource())
{
   // Използваме ресурса
}

// Ресурсът е освободен
```

Съществува обаче проблемът да забравите да извикате **Dispose()** на обекта (или клиент на вашата библиотека с ресурси да забрави да го направи) и тогава, в най-добрия случай ще изтече памет. На помощ идват финализаторите, които могат да ви гарантират това извикване на метода **Dispose()** (ако вече не е извикан).

IDisposable и Finalize

Винаги са двама, не повече, не по-малко:
майстор и ученик. ***Йода***

На практика съществуват няколко въпроса, на които трябва да си отговорим преди да пристъпим към имплементация на **IDisposable** и написването на финализатор:

Нуждае ли се класът ни от експлицитно ръчно управление?

Повечето начинаещи .NET програмисти (да не се бърка с начинаещ програмист), особено тези които имат предишен опит с езици, при които паметта се управлява ръчно (например C и C++) тайно се надяват

отговорът да е "да". Уви, както при много други въпроси, свързани с програмирането, отговора е "зависи", но в повечето случаи е "не".

Правилният въпрос е "Обвива ли класът ни неуправляван ресурс?".

По-напредналите .NET програмисти ще ви кажат, че ако класът ви използва неуправляван ресурс, задължително трябва да има финализатор и да имплементира **IDisposable**. Проблемът в подобно изказване е думичката "използва". Например, ако класът ви използва неуправляван ресурс, който вече е обвит в управляван клас, не само не е необходимо да създадете повторен клас за обвиването на ресурса, но това може да доведе и до проблеми, например:

```
class BadPractice
{
   ~BadPractice()
   {
      mEventLog.Close(); //или ((IDisposable)mEventLog).Dispose()
   }

   private EventLog mEventLog; // инициализиран някъде другаде
}
```

По време на извикване на финализатора на **BadPractice**, обектът, сочен от полето **mEventLog** вече може да е "събран" от garbage collector и дори може да се е изпълнил финализаторът му, при което ще получите изхвърляне на изключението **ObjectDisposedException**.

Ако заменим "използва" с "обвива", тогава ние също ви препоръчваме имплементирането и на финализатор и на интерфейса **IDisposable**.

Може ли да разчитаме на коректна употреба?

Разумно ли е да разчитаме, че класът ни ще бъде използван коректно и ако не, може ли да го защитим от неправилна употреба? Краткия отговор е "не и да". Не само, че не можете да разчитате на коректна употреба на вашите класове, а е почти гарантирано, че ще бъдат използвани неправилно, при това находчиво неправилно, от по-неопитни програмисти. Добрата новина е, че можете да защитите класовете си срещу неправилна употреба, а лошата е, че в тази глава ще ви покажем как да го направите, но само в контекста на имплементацията на интерфейса **IDisposable** и финализатора на класа.

Съществуват няколко начина за неправилна употреба на вашия клас:

- Потребителят на класа да забрави да извика **Dispose()**
- Потребителят на класа да извика **Dispose()** повече от един път
- Потребителят на класа да извика и **Dispose()** и **Finalize()** (евентуално повече от един път)

Ако се съмнявате във втория начин, просто си представете, че програмист е ползвал изрично извикване на **Dispose()**, по-късно се е научил да ползва оператора **using** и е забравил да премахне изричното извикване. Ако се съмнявате, че някой може да постигне третата неправилна употреба, се върнете на текста за финализация и по-специално на техниката "съживяване на обекти".

Трябва ли да напишем кода си безопасен за употреба от многонишкови приложения?

Този въпрос няма правилен отговор. Не може да контролираме по никакъв начин броя на нишките, използващи една и съща инстанция на нашия клас. Отговорът се диктува от дизайна, който сте избрали. Ако обаче сте избрали да поддържате извиквания към обекта ви от много нишки, тогава считайте, че трябва да поддържате това както от финализатора, така и от метода **Dispose()**. Освен това имайте предвид факта, че кодът в нишките може да бъде прекъснат в най-неподходящ за вас момент, а именно по време на освобождаване на ресурсите.

Примерна имплементация на базов клас, обвиващ неуправляван ресурс

Имплементирането на интерфейса **IDisposable** изглежда измамливо лесно, тъй като този интерфейс съдържа един единствен метод, който не приема параметри и не връща резултат. Същото се отнася и до т. нар. финализатор.

Написването на код, който да се използва като шаблон е добра практика, но написването на код, който е готов да бъда използван (и преизползван) без модификация е много по-удобно и води до по-малко грешки. Ето защо ви предлагаме имплементация на базов клас, който има следните предимства:

- Класът е завършен и единственото, което трябва да направите е да го наследите (това разбира се е и основният му недостатък)
- Защитен е от всички възможни грешки от неправилна употреба
- Поддържа многонишкови приложения
- Защитава обекта от неочаквано прекъсване на нишката, в която се изпълнява унищожаването на ресурса[10]

```
// Базов клас за обекти, обвиващи ресурс
public abstract class ResourceWrapperBase : IDisposable
{
   // Член-променливи и константи
```

[10] "Най-находчивите" от вас могат да опорочат защитата (помислете как)

```
private const int FALSE = 0;
private const int TRUE = 1;
private int mDisposed = FALSE;

~ResourceWrapperBase()
{
   DisposeImpl(false);
}

// Имплементация на интерфейса IDisposable
public void Dispose()
{
   DisposeImpl(true);
}

// Същинската имплементация е скрита в т. нар. шаблонен
// метод[11]
private void DisposeImpl(bool aDisposing)
{
   // Проверяваме дали обектът вече не е бил освободен и
   // веднага вдигаме флага, за да предотвратим паралелното
   // изпълнение на същия код от друга нишка
   if (Interlocked.CompareExchange(ref mDisposed,
      TRUE, FALSE) == TRUE)
   {
      return;
   }

   // Отваряме try...finally блок за да се предпазим
   // от възможността възникването на асинхронно изключение
   // да прекъсне кода за освобождаването на ресурсите
   // в неподходящ момент
   try
   {
      if (aDisposing)
      {
         // Експлицитно освобождаване: делегираме към наследника
         DisposeManagedResources();
      }

      // Отваряме try...catch блок за да предотвратим
      // възможността нашият код да предизвика изключение
      // по време на финализация
      try
      {
         // Делегираме към наследника
         DisposeUnmanagedResources();
```

[11] Виж шаблона template method в книгата "Шаблони за дизайн"

```
        }
        catch
        {
          if (aDisposing)
          {
            // Изхвърляме повторно изключението, ако
            // обектът не се финализира в момента
            throw;
          }
        }
      }
      finally
      {
        if (aDisposing)
        {
          // Съобщаваме на системата за почистване на боклука, че
          // обектът вече не се нуждае от финализация
          GC.SuppressFinalize(this);
        }
      }
    }

    // Функция за проверка дали обектът вече е освободен
    // предназначена да се ползва от наследниците на класа
    protected bool IsDisposed()
    {
      return Interlocked.CompareExchange(ref mDisposed,
        FALSE, FALSE) == FALSE;
    }

    // Помощна функция, предназначена за наследниците на класа,
    // която трябва да бъде извикана във всички не-private методи
    protected void EnsureNotDisposed()
    {
      if (IsDisposed())
      {
        throw new ObjectDisposedException(GetType().FullName);
      }
    }

    // Наследниците на този клас предефинират следните методи
    // като по този начин "попълват" шаблона на метода ни
    // Dispose(). Предоставили сме имплементация по подразбиране
    protected virtual void DisposeManagedResources() {}
    protected virtual void DisposeUnmanagedResources() {}
}
```

Въпреки, че кодът съдържа подробни коментари, сме длъжни да направим няколко разяснения, особено заради читателите, които нямат опит с многонишкови приложения.

- Унищожаването на ресурса е поместено в един общ метод (**`DisposeImpl(…)`**), който се извиква както от финализатора, така и от метода **`Dispose()`**. Методът **`DisposeImpl(…)`** приема като параметър флаг, указващ дали се извиква по време на финализация или изрично от потребителския код.
- Статичният метод **`CompareExchange(…)`** на класа **`Interlocked`** е атомарна операция[12] (т.е. такава, при която е гарантирано изпълнението ѝ само от една нишка, независимо от броя на нишките и процесорите в дадена система), която извършва следните действия:
  - Проверява дали стойността в първия аргумент е равна на третия аргумент
  - Ако стойността е равна, присвоява на първия аргумент стойността на втория аргумент
  - Връща първоначалната стойност на първия аргумент

  В случая кодът:

```
if (Interlocked.CompareExchange(ref disposed,
   TRUE, FALSE) == TRUE)
{
   return;
}
```

  е еквивалентен на следния код при приложение без поддръжка на много нишки:

```
if (disposed)
{
   return;
}
disposed = true;
```

- Влизането в първия **`try…finally`** блок е задължително за да се предпазим от прекъсване на нишката от извикването на методите **`Thread.Abort`** или **`Thread.Interrupt`**[13]

[12] Имплементацията на **`CompareExchange()`** използва специална инструкция с префикс на процесорите на Intel (**`lock cmpxchg`**) поради което работи много бързо

[13] Повече информация ще намерите в глава 16

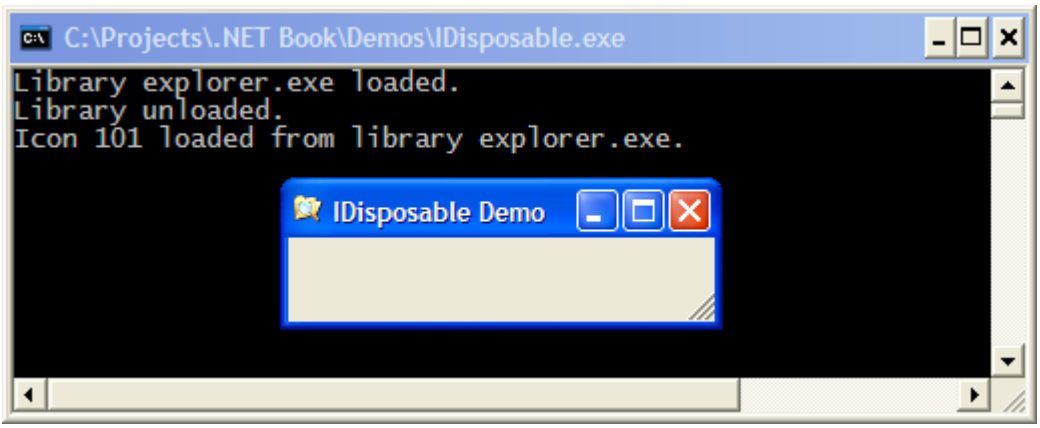
Обвиване на управляван ресурс – пример

За да оцените лекотата на използване на класа **ResourceWrapperBase** ще го приложим, за да обвием манипулатор на динамична библиотека и икона в следващия пример:

```
class WindowsLibrary : ResourceWrapperBase
{
   private IntPtr mLibraryHandle = IntPtr.Zero;

   public WindowsLibrary(string aFileName)
   {
      mLibraryHandle = LoadLibrary(aFileName);
      Console.WriteLine("Library {0} loaded.", aFileName);
   }

   public IntPtr Handle
   {
      get
      {
         return mLibraryHandle;
      }
   }

   protected override void DisposeUnmanagedResources()
   {
      if (mLibraryHandle != IntPtr.Zero)
      {
         FreeLibrary(mLibraryHandle);
         mLibraryHandle = IntPtr.Zero;
         Console.WriteLine("Library unloaded.");
      }
      base.DisposeUnmanagedResources();
   }

   [DllImport("kernel32.dll", SetLastError=true)]
   static extern IntPtr LoadLibrary(string lpFileName);

   [DllImport("kernel32.dll", SetLastError=true)]
   static extern int FreeLibrary(IntPtr hModule);
}

class WindowsIcon : ResourceWrapperBase
{
   private Icon mIcon;

   public WindowsIcon(string aFile, int aIconId)
   {
      using (WindowsLibrary lib = new WindowsLibrary(aFile))
      {
         IntPtr hIcon = LoadIcon(lib.Handle, aIconId);
```

```
            mIcon = Icon.FromHandle(hIcon);
        }
        Console.WriteLine("Icon {0} loaded from library {1}.",
            aIconId, aFileName);
    }

    protected override void DisposeManagedResources()
    {
        if (mIcon != null)
        {
            mIcon.Dispose();
            Console.WriteLine("Icon disposed.");
        }
    }

    public Icon Icon
    {
        get
        {
            return mIcon;
        }
    }

    [DllImport("user32.dll", SetLastError=true)]
    public static extern IntPtr LoadIcon(IntPtr hInstance,
        int lpIconId);
}

class IDisposableDemo
{
    static void Main()
    {
        const int FIRST_EXPLORER_ICON = 101;
        WindowsIcon explorerIcon =
            new WindowsIcon("explorer.exe", FIRST_EXPLORER_ICON);
        using (explorerIcon)
        {
            using (Form form = new Form())
            {
                form.Text = "IDisposable Demo";
                form.Icon = explorerIcon.Icon;
                form.ShowDialog();
            }
        }
        Console.WriteLine("End of Main() method.");
    }
}
```

При изпълнение на горния пример се получава следния резултат:



Вижда се, че примерът зарежда и използва неуправляваните ресурси "Windows библиотека" и "Windows икона" и ги освобождава експлицитно чрез **`using`** конструкцията. Благодарение на наследяването на класа **`ResourceWrapperBase`** дори ако тези ресурси не се освобождаваха експлицитно, те щяха да бъдат освободени от финализаторите на съответните им обвиващи класове.

Close() и експлицитна имплементация на IDisposable

В някои приложни области едни термини "звучат по-добре" от по-обобщените такива. Файловете, потоците и връзките към бази данни ги затваряме (Close), заключванията (lock) ги освобождаваме (Release) и т.н.

В едни от най-ползваните асемблита в .NET Framework 1.1 (**`System.dll`**, **`System.Data.dll`**, **`System.Drawing.dll`**, **`System.WindowsForms.dll`** и **`System.Web.dll`**) има общо 242 класове и/или структури, имплементиращи **`IDisposable`** от които 32 го имплементират експлицитно (става въпрос за версия 1.1 на .NET Framework).

Последните извикват (невидимия от Visual Studio IntelliSense метод) **`Dispose()`** от публичния си метод **`Close()`**.

Пример:

```
public class File : IDisposable
{
   // Експлицитна имплементация на интерфейса IDisposable
   void IDisposable.Dispose()
   {
      // ...
   }

   // Преобразуването към IDisposable е наложително
   public void Close()
   {
```

```
        ((IDisposable)this).Dispose();
    }
}
```

Класове като **File** могат да бъдат унищожени надеждно по два начина – като се обвие използването на обекта в **try…finally** блок или като се използва оператора **using** (в C#). На практика, болшинството от C# програмистите привикват към и предпочитат оператора **using**, ето защо написването на подобен на **Close()** метод от ваша страна лично ние считаме за излишно.

Кога да извикваме IDisposable.Dispose()?

На въпроса за освобождаването на кои ресурси трябва да се грижим ръчно чрез извикване на **IDisposable.Dispose()** няма прост отговор. Правилото е, че ръчно трябва да се освобождават всички класове, които обвиват неуправляван ресурс и имплементират **IDisposable**.

Например класът **FileStream** имплементира **IDisposable** и обвива неуправлявания ресурс "файл". Следователно той трябва да се освобождава ръчно чрез **using** конструкцията или чрез **try … finally** блок. Класът **StringReader** също имплементира **IDisposable**, но няма нужда да бъде освобождаван ръчно, защото не държи в себе си неуправляван ресурс.

Когато ползвате даден клас от .NET Framework или от друга библиотека, трябва да проверите дали той имплементира **IDisposable** и да помислите дали той обвива неуправляван ресурс. Ако се съмнявате, освобождавайте ресурса ръчно. Това няма да навреди.

Взаимодействие със системата за почистване на паметта

Намирам липсата ти на вяра обезпокоителна.
Дарт Вейдър

.NET Framework предлага средства за взаимодействие със системата за почистване на паметта (garbage collector). Взаимодействието се осъществява с помощта на статичните публични методи на класа **GC**, някои от които ще разгледаме в тази точка.

Почистване на паметта

Можем да предизвикаме стартирането на почистването на паметта с извикването на метода **GC.Collect()**. Извикването на този метод без параметри предизвиква пълно почистване на всички поколения памет. Извикването на overload варианта на същия метод с аргумент номер на поколение, предизвиква почистване на всички поколения, започвайки от 0 до указаното.

Помогнете на GC като не й помагате

Като правило (което както ще видите по-нататък си има изключения) се старайте да не помагате на системата за почистване на паметта. Тя е "произведение на изкуството", внимателно проектирана така, че да гарантира висока ефективност при различен род приложения. Освен това, GC се самонастройва като следи поведението на заделяне на памет на вашето приложение.

Ако грижливо проектирате приложението си, няма да има нужда да мислите за GC, но ако си мислите, че се нуждаете да предизвикате почистване, значи нещо се е объркало. Трябва да се запитате какво сте направили, че е нужно да предизвикате почистване и от какво точно почистване се нуждаете – на поколение 0, 1 или 2?

Ако точно сега се нуждаете от почистване на поколение 0

Почистването на поколение 0 се случва достатъчно често и е сравнително "евтино". GC използва темпото с което заделяте памет и големината на кеша на процесорите ви за да определи колко памет да ви позволи да заделите преди да стане изгодно да почисти поколение 0. Ако принудите GC да почисти паметта преди настъпването на този момент е възможно да му дадете прекалено малко времеви интервал, за да определи размера на заделената памет, необходима за следващото почистване и в крайна сметка да се окажете с повече почиствания на поколение 0 от колкото се нуждаете. Тъй като размерът на паметта на поколение 0 така или иначе не става прекалено голям, най-добре е да оставите GC да извършва автоматично почистването, както прецени за най-добре. Ако наблюдавате средно 1 почистване на поколение 0 в секунда, всичко е наред.

Ако точно сега се нуждаете от почистване на поколение 1

Първият проблем е, че `GC.Collect()` не обещава почистване на обекти от поколение 1. Следващия проблем е, че за да знаете размера на поколение 1 (иначе за какво ви е да го почиствате) трябва да наблюдавате темпото на оцеляване на обектите от поколение 0, така че в крайна сметка е доста сложно да разберете дали наистина се нуждаете от почистване на поколение 1. Последният проблем е, че поколение 1 е също сравнително "евтино" за почистване от GC (въпреки, че е по-скъпо от поколение 0, тъй като го включва в себе си) и отново е безсмислено в повечето случаи да предизвикате GC да го почисти.

Без да се впускаме в повече подробности, ако се нуждаете от почистване на поколение 1, не го предизвиквайте. Вместо това вижте дали не може да промените кода или алгоритмите, които използвате, така че да направите обектите недостижими колкото е възможно по-бързо. Вашата цел е да направите по-дълго живеещите обекти да станат със средна продължителност на живот, а последните с кратка, след което може да спрете да се тревожите за поколение 1. След като обектите ви вече са в поколение 0, както сами сте се уверили, няма нужда да предизвиквате GC изобщо. Ако

наблюдавате едно почистване на поколение 1 средно на десетина секунди, всичко е наред.

Ако точно сега се нуждаете от почистване на поколение 2

Почистването на поколение 2 означава цялостно почистване на паметта, следователно е значително по-скъпо от това на поколения 1 и 2. Отново, имайте предвид, че **`GC.Collect(2)`** не ви обещава почистване на поколение 2. Ако си мислите, че имате нужда от почистване на поколение 2, значи дизайна на приложението ви се нуждае от щателен преглед.

Ако наблюдавате почистване на поколение 2 на 100 секунди, всичко е наред.

Кога може да помогнете на GC?

Има смисъл да извикате **`GC.Collect()`** ако някое, неповтарящо се често събитие се е случило току-що и това събитие е допринесло за смъртта на много стари обекти.

Класически пример за това е ако пишете desktop приложение и предоставите на потребителя голяма и сложна форма, асоциирана с много данни в нея. Потребителят е създал с помощта на тази форма голям XML или един или повече **`DataSet`** обекта. Когато формата се затвори, тези обекти са мъртви и **`GC.Collect()`** ще освободи паметта им.

Въпреки, че системата за почистване на паметта е самонастройваща, тя не може да предвиди абсолютно всеки шаблон за заделяне на памет и в горния случай най-вероятно няма да успее да предвиди, че умират много обекти от поколение 2 (обектите от поколение 0 и 1 ще са с голяма вероятност вече почистени). Само обектите, които първоначално са свързани с формата ще преминат в поколение 2. В този момент е много добре от гледна точка на производителността да се извика почистване на поколение 2.

Така че, когато неповтарящо събитие, включващо смъртта на много обекти се случи (например при завършване на инициализация на приложението или при затварянето на голям диалогов прозорец) може да вмъкнете **`GC.Collect()`** за да освободите паметта. Не го правете, ако обектите не са много.

Имайте предвид, че Microsoft настройват системата за почистване на паметта все повече и повече и в следващата й версия е възможно вашите извиквания към **`GC.Collect()`** да попречат на GC да работи ефективно вместо да помогнат. Ето защо е добра идея да ги обгърнете в условен метод, например:

```
#define HELP_GC

public sealed class GCHelper
{
```

```
   [Conditional("HELP_GC")]
   public static void Collect()
   {
      GC.Collect();
   }
   private GCHelper() {}
}
```

В примера се използва атрибутът **System.Diagnostics.Conditional**, с който се указва, че методът **Collect()** на класа **GCHelper** е условен метод и съществува само ако е дефиниран символът **HELP_GC** по време на компилация. В противен случай методът изчезва заедно с всички извиквания към него.

Финализаторите увеличават живота на обектите

В C++ е общоприето да се използват деструктори, за да се освобождава памет или по-общо ресурси. Както вече знаете, финализаторите не са деструктори, така че като изключим обгръщането на неуправляван ресурс, вашите класове не се нуждаят от финализатор. GC ще почисти боклука от членовете на вашия "мъртъв" обект без да е нужно да им зададете стойност **null**. Има смисъл да присвоите **null** на част от членовете на типа си, само ако искате те да бъдат почистени докато обектът ви е все още "жив".

Дефинирайте финализатор само ако класът ви обгръща неуправляван ресурс!

Ако случая не е такъв, вие просто не се нуждаете от финализатор. Добавянето на такъв със сигурност изпраща обектите от вашия клас в по-горно поколение и увеличава работата на системата за почистване на паметта.

Имплементирайте IDisposable без финализатор

Когато знаете, че този, който ще ползва класа сте вие, и че няма да забравите да извикате **Dispose()** добавете финализатор само за DEBUG компилация за да сте сигурни, че викате **Dispose()** навсякъде, където е необходимо:

```
class SomeDisposable : IDisposable
{
#if DEBUG // Финализаторът съществува само в DEBUG build
   ~SomeDisposable()
   {
      Debug.Assert(mDisposed, "Dispose wasn't called!");
   }
#endif
   public void Dispose()
```

```
    {
        // ... имплементация
        mDisposed = true;
        GC.SuppressFinalize(this); // виж по-долу
    }

    private bool mDisposed = false;
}
```

За съжаление, горната техника не носи информация кой и къде е забравил да извика **Dispose()**. Ако обаче ползвате редовно оператора **using**, няма да ви се наложи да търсите виновния код.

Потискане на финализацията

Както вече видяхте в примерната имплементация на базов клас, предоставящ финализатор и имплементация на **IDisposable**, след извикването на метода **Dispose()** е добре (макар и незадължително) да потиснете финализацията като оптимизация. Потискането на финализация се извършва с помощта на метода **GC.SuppressFinalize(…)**, който приема като параметър инстанция на тип. Цената за извикване на този метод е просто промяната на 1 бит в заглавната част на обекта.

Изчакване до приключване на финализацията

Ако имате случай, в който сте помогнали на GC, като сте извикали **GC.Collect()**, можете да помогнете още малко, като извикате метода **GC.WaitForPendingFinalizers()**. Извикването на този метод ще принуди GC да обработи всички финализатори на маркираните за финализация обекти от извикването на **GC.Collect()**. Добрата новина е, че по този начин неуправляваните ресурси, обвити в почистените обекти ще бъдат унищожени (ресурсите също заемат памет). Лошата е, че ще ви е необходимо повече време за да приключи финализацията им. Общо взето, ако е настъпил добър момент за предизвикването на пълно почистване, вие би трябвало да сте склонни да отделите това време.

Регистриране на обекта за финализация

Ако поради някаква причина сте премахнали обекта си от опашката за финализиране (извиквайки **GC.SuppressFinalize(…)**), може да го добавите отново, като извикате **GC.ReRegisterForFinalize(…)**. Единствената смислена употреба на този метод е да съживите обекта, който се финализира в момента, извиквайки **GC.ReRegisterForFinalize(this)**. Това може да се наложи да направите ако по време на финализация класът ви не успее да се финализира успешно и има нужда да опита пак след известно време.

Определяне поколението на обект

В случай, че сте решили да предизвикате почистване на паметта, може да определите в кое поколение се намират обектите, които искате да почистите, за да извикате **GC.Collect()** до това поколение. Извикването на **GC.GetGeneration(object)** ви връща поколението на обекта. Ако например сте решили да почистите обект и обектите от неговото поколение (и надолу до поколение 0), използвайте следния код:

```
public sealed class GCHelper
{
   public static void CollectObjectGeneration(object obj)
   {
      if (obj != null)
      {
         GC.Collect(GC.GetGeneration(obj));
      }
   }
}
```

Максималното поколение в даден момент може да получите като извикате свойството **GC.MaxGeneration()**.

Pinning

Английският глагол pin означава забождам, приковавам, притискам (обикновено с топлийка/карфица). В контекста на взаимодействие със системата за почистване на паметта, забождането на обект означава да не позволите на GC за известно време да мести обекта на друго място в паметта (което обикновено се случва при събиране на боклука на поколението, в което "живее" обекта). В този текст ще използваме думата pinning.

Pinning прилича малко на финализацията по това, че и двете съществуват, защото ни се налага да работим с неуправляван (native) код.

Кога е необходим pinning?

Обектите се pin-ват по три причини:

- при създаване на инстанция на класа **GCHandle** с тип **GCHandleType.Pinned** (което едва ли ще ви се наложи да използвате);
- при използване на ключовата дума **fixed** в C# (или **__pin** в Managed C++);
- по време на взаимодействие с неуправляван код (Interop), някои аргументи се pin-ват от Interop (например, за да се подаде обект **String** като **LPWSTR**, Interop pin-ва буфера по време на изпълнението извиканата функция).

За обектите от малкия хийп (поколения 0, 1 и 2), техниката "pinning" е единственият начин потребителят да успее да фрагментира хийпа.

За блока от памет за големи обекти (LOH), pinning в момента е нулева операция, тъй като в настоящата имплементация на Garbage Collector, обектите в LOH не се пренареждат, както при поколения от 0 до 2. Разбира се, това е имплементационен детайл, на който не бива да разчитате.

Използвайте pinning внимателно

Фрагментирането на хийпа е лошо. То кара GC да работи по-усърдно, като вместо просто да "прецежда" достъпните обекти, сега трябва да запомни кои "живи" обекти са pin-нати и да се опитва да вмъква обекти в свободните места между pin-натите обекти.

Когато ви се налага да pin-нете обект, имайте предвид следното:

- Ако го направите за кратко време, операцията е "евтина". Как може да прецените какво е кратко време? Ако по време на pinning не се случва събиране на боклука, операцията просто вдига бит в заглавната част на обекта и след приключването си го сваля. Но ако по това време се задейства GC "забодените" обекти не трябва да се местят. Следователно "кратко време" е времето, през което GC не забелязва, че обект е pin-нат. Това означава, че докато pin-вате обекти не трябва да се случват никакви или почти никакви заделяния на памет (които иначе биха могли да предизвикат нуждата от почистване на боклук).
- Ако все пак често ви се налага да работите с буфери, които да pin-нете, преди да ги подадете като параметри на Interop функции например, можете да създадете пул от буфери и да предизвикате GC така, че обектите да минат в поколение 2. Тъй като обектите от поколение 2 се пренареждат доста по-рядко (а тези в LOH изобщо не се пренареждат), ще нанесете много по-малка "вреда" на GC.

Удължаване живота на променливите при Interop

Нека имаме клас, който обвива манипулатор към неуправляван ресурс и също така метод, който връща друг манипулатор, използвайки първия:

```
class ResourceWrapper : IDisposable
{
   IntPtr hRes;
   public IntPtr Method()
   {
      return SomeInteropFunction(hRes);
   }
   ~ResourceWrapper() { … }
}
```

Нека сега си представим, че трябва да извикаме функция чрез Interop, която приема такъв манипулатор като параметър:

```
public void SomeMethod
{
   using (ResourceWrapper rw = new ResourceWrapper())
   {
      PInvokeHelper.InvokeLibFunction(rw.Method())
   }
}
```

На пръв поглед нещата изглеждат добре и вие може би си мислите, че променливата **rw** е "жива" до затварящата скоба на оператор **using**. Грешката е в това, че преди да се извика **InvokeLibFunction** се изчисляват нейните параметри, а именно манипулаторът, който очаква. Ето защо кодът в действителност би изглеждал така:

```
public void SomeMethod
{
   using (ResourceWrapper rw = new ResourceWrapper())
   {
      IntPtr h = rw.Method();
      PInvokeHelper.InvokeLibFunction(h);
   }
}
```

Нека не забравяме, че всъщност операторът **using** е само "синтактична захар" и кодът в действителност е нещо такова:

```
public void SomeMethod
{
   ResourceWrapper rw = new ResourceWrapper();
   try
   {
      IntPtr h = rw.Method();
      PInvokeHelper.InvokeLibFunction(h);
   }
   finally
   {
      ((IDisposable)rw).Dispose();
   }
}
```

Естествено компилаторът, както и JIT компилаторът могат да преценят, че тъй като резултатът от извикването на **rw.Method()** е прост стойностен тип (**IntPtr** просто обвива един **int**), референцията към **rw** след извикването на този метод е ненужна, следователно готова за събиране от GC (това нямаше да се случи, ако просто връщахме **hRes** от метода, но ние връщаме нов манипулатор, към който класът **ResourceWrapper** няма референция). Кодът в този случай би могъл да се пренареди по следния начин:

```
public void SomeMethod
{
   ResourceWrapper rw = new ResourceWrapper();
   IntPtr h = IntPtr.Zero;
   try
   {
      h = rw.Method();
   }
   finally
   {
      ((IDisposable)rw).Dispose();
   }
   // тук обектът вече е унищожен
   PInvokeHelper.InvokeLibFunction(h);
}
```

Тъй като **rw** може да бъде унищожен, манипулаторът, върнат от извикването на **Method** е невалиден (ако например вторият манипулатор зависи от първия).

Как така е възможно някой да си помисли, че обектът е готов за почистване? Ами много просто – извикването на **Method()** връща манипулатор, към който никой няма референция. Още в този момент обектът е готов за GC, тъй като кодът по-долу не го използва, освен, за да се извика **Dispose()**. Ако не използвахме операторът **using**, обектът би могъл да бъде почистен веднага след напускането на метода **Method()**.

Естествено подобни спекулации може да ви накарат да настръхнете и вероятно вече си мислите, че дори да не изпаднете в точно тази ситуация, може неволно да напишете код, който да я предизвика. Ситуации като горната, обаче обикновено са свързани с Interop, с който не се сблъсквате често, а освен това съществува решение на проблема.

Статичният метод **GC.KeepAlive(object)** приема обект като параметър и служи като индикация за компилатора, JIT компилатора и най-вече GC да не събира обекта до момента, в който се извика **KeepAlive()**. Първоначалният код с вмъкнат **GC.KeepAlive(…)** би изглеждал така:

```
public void SomeMethod
{
   using (ResourceWrapper rw = new ResourceWrapper())
   {
      PInvokeHelper.InvokeLibFunction(rw.Method())
      // може да бъдем сигурни, че InvokeLibFunction ще бъде
      // извикана преди rw да бъде почистен
      GC.KeepAlive(rw);
   }
}
```

Слаби референции

Слабите референции (weak reference) представляват референция към обект, която подлежи на почистване от системата за почистване на боклук след като всички силни референции към обекта отпаднат. Имайте предвид, че при недостиг на памет първо се почистват обектите сочени от слаби референции.

Създаване на слаба референция

Референция към достижим обект (обект с корен) се нарича силна референция (strong reference). Силна референция може да се превърне в слаба, като се създаде инстанция на класа **`System.WeakReference`** и се подаде силната референция като параметър на конструктора. Обаче само конструирането на **`WeakReference`** обект не прави силната референция слаба. За целта, на всички корени, сочещи обекта, трябва да се присвои **`null`**. Ето един пример:

```
// Създаваме нов обект и го присвояваме на променлива.
// Това създава силна референция.
object obj = new object();

// Създаваме слаба референция към обекта
WeakReference wr = new WeakReference(obj);

// Тук все още имаме силна референция към обекта. Премахваме я.
obj = null;
```

Получаване на силна референция от слаба

Слабата референция също сочи към достижим обект, наричан цел (target) и може да се превърне отново в силна референция като се присвои стойността от свойството **`Target`** на съответната променлива. Свойството **`IsAlive`** показва дали обектът вече не е почистен.

```
if (wr.IsAlive)
{
   // Обектът още не е почистен, създаваме силна референция
   object obj = wr.Target;
}
else
{
   // Обекта вече е почистен от GC, wr.Target е null
}
```

Сценарии за употреба на слаби референции

Слабите референции се използват за съхранение на данни, които не са критични за приложението. Под некритични имаме предвид, че приложе-

нието не се нуждае от тези данни за константно време и може да изчака малко време за тяхното пресъздаване.

Например едно приложение може да обхожда твърдия диск и да прави списък с всички файлове, открити във всички директории и техните поддиректории. Този списък би могъл да заема доста памет. Създаването на този списък е бавно, но ако той липсва, винаги може да бъде създаден отново (с цената на повторно обхождане на всички директории по диска). За съхранението на списъка може да се използва слаба референция. Така когато списъкът потрябва и слабата референция не го е освободила поради недостиг на памет, списъкът ще може да се използва директно. В противен случай ще трябва да се изгради отново.

Имайте предвид, че една слаба референция има управлявана и машиннозависима (native) част. Управляваната част е самият клас **`WeakReference`**. В конструктора си той създава GC манипулатор (което е native частта) и вмъква запис в таблицата за манипулаторите на домейна на приложението си (AppDomain). Обектът, към който сочи слабата референция ще умре, когато няма силни референции към него, а също и самата слаба референция, когато няма силни референции към нея (все пак тя също е управляван обект).

Слабата референция съдържа манипулатор с големината на 1 указател (32 бита на 32-битови архитектури), едно булево поле и GC манипулатора, който също е с големината на 1 указател, така че ако имате много малък обект, да кажем съдържащ 1 int поле, вашият обект ще изразходва 12 байта памет (размера на минималния обект). Същият обект, вмъкнат в **`WeakReference`**, ще харчи поне още 9 байта. Следователно не си създавайте сами ситуации в които създавате много слаби референции, сочещи малки обекти.

Ефективно използване на паметта

Люк: Ще опитам.

Йода: Не. Не опитвай. Направи го. Или не. Няма опитване.

Едно от основните разлики между управлявания и неуправлявания код е автоматичното управление на паметта. Вие заделяте нови обекти, но garbage collector автоматично ги освобождава за вас когато вече не се използват. GC се изпълнява от време на време, често незабележимо, като най-често спира вашето приложение само за милисекунда или по-рядко две или повече.

Ако следвате съветите в тази тема, общата цена на използването на garbage collector ще бъде незабележима, конкурентна или дори по-добра от традиционните в C++ **`new`** и **`delete`**. Амортизираната цена на създаване и на по-късното освобождаване на обект е достатъчно ниска, че да е възможно да създавате десетки милиони малки обекти в секунда.

Системата на .NET за почистване на паметта предоставя изключително бързо заделяне на памет без дългосрочни проблеми с фрагментацията, но е възможно да пишете код, който да доведе до по-малка от оптималната й производителност.

За да постигнете най-доброто, използвайте следните утвърдени практики:

Внимавайте с абстракциите

.NET Framework скрива толкова много детайли, че болшинството от програмистите без предишен опит с езици от по-ниско ниво нямат почти никаква представа за цената на техния код.

Може да заредите 1 мегабайт XML от уеб сайт с няколко реда код, нали? Толкова е лесно! Наистина. Толкова е лесно да похарчите мегабайти памет, докато зареждате XML данните само за да използвате няколко елемента от тях. В C или C++ е толкова "болезнено", че щеше да се позамислите и да проектирате или използвате API с push (SAX) или pull (**`XmlReader`**) модел. В .NET Framework просто можете да заредите целия XML на един раз. Може би го правите отново и отново. После може би вашето приложение не изглежда вече толкова бързо. Може би трябваше да помислите за цената на тези лесни за използване методи...

Изберете най-доброто за целта API или алгоритъм

Нека си представим, че трябва да напишем проста конзолна програма, която отпечатва последните N реда от даден текстов файл, посочен на командния ред[14]. Може да напишете програмата по много начини:

- Заделяте масив с N елемента и докато четете файла ред по ред, попълвате масива като цикличен буфер (това е добро решение по отношение заделянето на памет, но не толкова добро по отношение на скоростта, ако файлът е голям, а освен това за всеки ред ще създадете нова инстанция на класа **`System.String`**).
- Прочитате файла на един дъх (с помощта на метода **`ReadToEnd()`** на класа **`StreamReader`**) и го сканирате отзад напред, отброявайки броя на достигнатите знаци за нов ред, докато достигнете до N (това решение е лошо по отношение на използване на паметта, но дава идеята за следващото решение).
- Отваряте файла в двоичен режим и четете отзад напред докато преброите N реда или не достигнете началото на файла, като или
  - поставяте редовете отзад напред в предварително заделен масив от N елемента (жертвайки памет за сметка на скоростта) или
  - препрочитате файла от достигнатата позиция до края му, отпечатвайки всеки ред (жертвайки скорост за сметка на паметта).

[14] Такава програма има в Unix (и подобните й ОС) и се нарича **`tail`**

Ако знаете, че един ред не надминава например 80 символа, може да прочетете наведнъж N * 80 символа в един буфер, да затворите файла и да преминете през буфера, отпечатвайки всеки ред.

Поуката от примера е, че може да решите конкретна задача по много различни начини и обикновено при всеки от тях се налага да вземете важното решение дали да пожертвате повече памет за сметка на скоростта или обратно. Универсална рецепта няма и решението зависи от нуждите на вашето приложение.

Няма безплатен обяд

Почистването на обектите от GC, особено на тези от поколение 0 е много бързо, но не е "безплатно", дори ако голяма част от обектите са "мъртви". За да се открият (и маркират) живите обекти първо трябва да се приспят нишките и да се обходят техните стекове и други структури, за да се съберат коренните референции към обекти в хийпа.

Заделянето на памет за обект също не е безплатно. Обектите заемат място. Неумереното създаване на обекти води до по-често задействане на GC. Дори по-лошо, ненужното задържане на референции към безполезни графи от обекти ги поддържа "живи".

Може да срещнете скромни програмки с печални working sets от по над 100 MB, чиито автори отричат тяхната вина и вместо това присвояват лошата производителност на някакъв мистериозен, неразгадаем (и следователно нерешим) проблем, свързан със самия управляван код. Трагично! Обикновено в такива случаи след час изучаване на проблема с CLR Profiler и промяна на няколко реда код, програмите намаляват изискването си за динамична памет с повече от 10 пъти. Ако имате проблем с големината на вашия working set, първата ви стъпка трябва да бъде да погледнете работата в паметта във вашето приложение.

Не създавайте обекти без да е необходимо

Програмистите често пъти неволно създават повече и повече обекти просто защото автоматичното управление на паметта решава много заплетени проблеми и грешки при заделяне и освобождаване на обекти и просто защото е бързо и удобно.

Типичен пример за безсмислен разход на памет е проектирането на т. нар. *one-shot* класове. Такъв клас има един или повече конструктора, приемащи всички необходими аргументи за да се извърши някаква операция и един метод, изпълняващ операция нуждаеща се от тези аргументи. Такива класове заемат излишна памет вместо да реализират функционалността си като статичен метод. Повечето от вас ще разпознаят шаблона за дизайн ***Command***. Той има големи предимства, когато се използва разумно и при необходимост, но за съжаление, по-неопитните програмисти го използват по-често от необходимото.

Пример за one-shot клас, проектиран безсмислено да бъде инстанциран за да бъде използван:

```
class FileDownloader
{
   private readonly string mUrl;
   private readonly string mPath;

   public FileDownloader(string aFileUrl, string aLocalPath)
   {
      mUrl = aFileUrl;
      mUath = aLocalPath;
   }

   public void Download()
   {
      // сваляме файла от mUrl и го записваме в директория mPath
      // …
   }
}
```

Същата функционалност може да бъде проектирана като статичен метод, който не изисква създаването на излишен обект.

Пример:

```
class FileDownloadHelper
{
   public static void Download(string aUrl, string aPath)
   {
      // сваляме файла от aUrl и го записваме в директория aPath
      // …
   }
}
```

Ако по-късно решите, че се нуждаете от клас, който капсулира параметрите на заявката, с цел ползването му в шаблона за дизайн ***Command***, винаги може да напишете клас **`FileDownloadCommand`**, който делегира към **`FileDownloadHelper`**.

Ако искате да пишете наистина бърз управляван код, създавайте обектите с мисъл и само когато е необходимо. Това се отнася още повече за онези от вас, които проектират програмни интерфейси (APIs). Възможно е да проектирате тип и неговите методи така, че да изисква от клиентите да създават и освобождават обекти непрекъснато. Не го правете!

Създавайте обекти където е необходимо

Въпреки, че компилаторът на вашия език и JIT компилаторът по време на изпълнение правят оптимизации, те не са перфектни[15]. Долният пример

[15] JIT компилаторът няма достатъчно време за да прави агресивни оптимизации, каквито може да направи компилаторът

показва как може да напишете код, който се изпълнява 5 и повече пъти по-бавно просто от нехайство:

```
for (int i = 0; i < 5000; ++i)
{
   int buffer[] = new int[65536];
   // Правим някакво изчисление с buffer
}
```

Същият код може да се оптимизира значително, като просто се извади заделянето на памет преди цикъла:

```
int buffer[] = new int[65536];
for (int i = 0; i < 5000; ++i)
{
   // Правим някакво изчисление с buffer
}
```

Не създавайте обекти с излишни полета

Това, че на пръв поглед ви изглежда нормално да направите общ клас за възлите и листата на едно дърво, не означава, че трябва да имплементирате структурата от данни дърво по този начин. Ето типична имплементация на дърво, срещана в огромен брой проекти:

```
public sealed class LameTreeNode : IEnumerable
{
   private string mName;
   private ArrayList mChildren;

   public LameTreeNode(string aName)
   {
      mName = aName;
      mChildren = new ArrayList();
   }

   public string Name
   {
      get
      {
         return mName;
      }
   }

   public int Count
   {
      get
      {
         return mChildren.Count;
      }
```

```
   }

   public IEnumerator GetEnumerator()
   {
      return mChildren.GetEnumerator();
   }

   public void Add(TreeNode aChild)
   {
      mChildren.Add(aChild);
   }
}
```

Големите проблеми тук са поне два:

- Конструкторът без параметри на **`ArrayList`** създава по подразбиране масив от 16 елемента – похабена памет за създаването на масива. Ако елементът е стойностен тип, всички стойности ще бъде опаковани, което допълнително увеличава изискването за памет.
- Не се знае дали на възела ще бъдат добавени листа, т.е. дали той самият няма да остане листо, но паметта за масива се харчи във всички случаи. Това, разбира се, опростява имплементацията на **`Count`**, **`GetEnumerator()`** и **`Add(…)`**, но цената не е малка.

Същият клас може да се преправи така, че да изразходва по-малко допълнителна памет за елементите, които са листа:

```
public sealed class DecentTreeNode : IEnumerable
{
   private string mName;
   private ArrayList mChildren;
   private static IEnumerator mNullEnumerator =
       (new TreeNode[0]).GetEnumerator();
   private const int DEFAULT_CAPACITY = 4;

   public DecentTreeNode(string aName)
   {
      mName = aName;
   }

   public string Name
   {
      get
      {
         return mName;
      }
   }

   public int Count
   {
```

```
      get
      {
         return mChildren == null ? 0 : mChildren.Count;
      }
   }

   public IEnumerator GetEnumerator()
   {
      return mChildren == null ?
         mNullEnumerator :
         mChildren.GetEnumerator();
   }

   public void Add(TreeNode aChild)
   {
      if (mChildren == null)
      {
         mChildren = new ArrayList(DEFAULT_CAPACITY);
      }
      mChildren.Add(aChild);
   }
}
```

Както винаги, класът може да се подобри поне по още два начина:

- Ако знаете минимумът и/или максимумът на броя на листата на възела, можете да инициализирате **`ArrayList`** член-променливата с капацитет, подаден в конструктора на **`DecentTreeNode`**.
- Можете да създадете абстрактен клас **`Node`** и конкретни класове за възли и листа. В този случай се подразбира, че знаете кога ще създавате обект от единия или другия тип.

Не инициализирайте полетата в конструкторите

След като паметта за новосъздаден обект се задели, CLR го инициализира (конструира). CLR гарантира, че всички референтни полета са предварително инициализирани с **`null`** и всички примитивни скаларни полета са инициализирани с **`0`**, **`0.0`**, **`false`** или съответната нулева стойност. Следователно е ненужно повторно да ги инициализирате в дефинираните от вас конструктори. Длъжни сме да ви предупредим, че в текущата си имплементация компилаторът не оптимизира и не премахва повторни инициализации от конструкторите ви.

Не проектирайте излишно дълбоки йерархии

Вторият принцип на обектно-ориентираният дизайн гласи[16]:

[16] вж. "Шаблони за дизайн", изд. SoftPress

Предпочитайте композицията на обекти пред наследяването на клас.

Голяма част от програмистите обаче (особено тези с предишен опит с езици, които не поддържат наследяване), злоупотребяват с наследяването, веднъж след като придобият някакъв опит с ООП.

След като CLR задели памет за даден обект и инициализира полетата му, се извиква конструкторът на съответния тип. Конструкторът на всеки тип, дефиниран от програмиста или компилатора, първо извиква конструктора на базовия си тип, после изпълнява дефинираната от програмиста инициализация.

На теория, това може да бъде скъпо като време на изпълнение, тъй като ако имаме клас E наследяващ D, наследяващ C, наследяващ B, наследяващ A (наследяващ **`System.Object`**), тогава конструирането на E ще предизвика пет извиквания на конструктори. На практика нещата не са толкова зле, тъй като компилаторът слива в едно (inline) извикванията към празни конструктори на базовите класове.

Кеширани и некеширани ресурси

На места в .NET Framework стратегията за управляване на ресурси не е добре замислена. Да вземем например следните два реда код:

```
Brush brush = Brushes.White;
Font font = SystemInformation.MenuFont;
```

За първия ред не се изисква да извикате **`brush.Dispose()`**, тъй като колекцията **`Brushes`** ви връща кеширано копие. На втория ред, обаче има проблем, тъй като **`font`** обектът, върнат от свойството **`MenuFont`** е новосъздаден и като такъв трябва да му извикате метода **`Dispose()`** след като приключите работата си с него.

Очевидно ли е това от кода? Не. Споменато ли е някъде в документацията? Не. Внимавайте! Ако не сте сигурни, проверявайте с Reflector. Използвайте следното лесно за запомняне правило, когато проектирате вашите типове:

Ако инстанцията на обекта, който връщате, е кеширана от вас, използвайте свойство за извличането й. Ако създавате и връщате нова инстанция, използвайте метод. Именувайте методите и свойствата по начин, който не оставя съмнение за това дали върнатият обект е новосъздаден и се нуждае от освобождаване.

Ето един пример:

```
class Brushes
{
  public static Brush CreateSolidBlackBrush()
```

```
   {
      // Отговорността за освобождаване на ресурса е на
      // извикващия този метод
      return new SolidBrush(Color.Black);
   }

   public static Brush CachedBlack
   {
      // Отговорността за освобождаване на ресурса е на
      // програмиста, проектирал класа Brushes
      return mCachedBlackBrush;
   }
}
```

Заделете цялата памет, нужна за създаването на структура от данни, наведнъж

Напишете програма, която създава масив от 1 милион **int** елемента и прост свързан списък от 1 милион възли, като всеки възел обвива един **int** елемент. После измерете времето, нужно да съберете първите хиляда, 10 хиляди, 100 хиляди и 1 милион елемента. Повторете всеки цикъл много пъти (вкарайте го във външен цикъл) за да измерите скоростта.

Указва се, че колкото повече данни обхождате, толкова по-бавно се държи свързаният списък. Версията с масива е винаги по-бърза, въпреки, че изпълнява два пъти повече инструкции. За 100 хиляди елемента, версията с масива е до 7 пъти по-бърза.

Защо? Първо, много по-малко възли се помещават в който и да е кеш на процесора. Всички заглавни части на обектите (8 байта) и връзките към следващия елемент (4 байта на 32-битова машина) заемат ненужно място. Вярно, че с версията със свързания списък заемате памет, само когато ви е нужна, но:

- Версията с масива заема 4 пъти по-малко памет (само 4 байта за 1 елемент, вместо 8 (заглавна част) + 4 (**int** числото) + 4 (връзката към следващия елемент) = 16 байта) и съответно по-голям брой елементи се помещават в кеша на процесора, а той е многократно по-бърз от коя да друга е памет.
- Днешните модерни процесори могат както да изпълняват инструкциите не в реда както са им подадени, така и да се опитват да прочитат данни преди да сме им ги поискали. В нашия случай, процесорът може да прочете още данни, преди да сме ги поискали за масива си.

Версията със свързания списък е винаги по-бавна, тъй като преди възела да достигне кеша на процесора, той не може да прочете следващия възел, т.е. имаме четене само на 1 възел в даден момент.

Каква е поуката тук?

Имайте предвид кеша на процесора в дизайна си

- Експериментирайте и премервайте. Трудно е, както да се предвидят страничните ефекти във всяка една програма, така и да се дадат някакви специфични указания как да проектирате най-ефективно структурите от данни, които ползвате.
- Близките данни се достъпват по-бързо, предпочитайте масиви пред свързани списъци.
- Когато масивите не ви вършат работа, използвайте хибридни структури, например списъци от по-малки масиви, масиви от масиви и т.н.
- Тъй като garbage collector запазва относителния ред на обектите, тези обекти, които за създадени заедно по едно и също време (и на същата нишка) обикновено остават заедно (и близо един до друг) в паметта. Използвайки това знание можете внимателно да създадете обекти, които ще се използват заедно, така, че да споделят общо място в кеша на процесора.
- Можете да разделите обектите си на "топли" и "студени" части, като топлите съдържат често използваните данни, а студените – рядко използваните – и могат да влизат и излизат от кеша на процесора без това да е осезаемо за приложението ви.

Използвайте следния модел за цената на пространството

- Размерът на стойностните типове обикновено е общият размер на всичките му полета, като полетата, които са по-малки от 4 байта се подравняват до 4 байта.
- Можете да имплементирате обединения на последователни полета (unions), като използвате атрибутите **`[StructLayout(LayoutKind.Explicit)]`** и **`[FieldOffset(n)]`**.
- Размерът на референтните типове е 8 байта (размерът на заглавната част на всеки референтен обект) + размера на всичките им полета, подравнен до 4-байтова стойност, като по-малките от 4 байта полета се подравняват.
- В C# при декларация на изброим тип може да укажете произволен целочислен тип, така че е възможно да дефинирате 8, 16, 32 и 64-битови изброими типове.
- Както и в C/C++ винаги може да изцедите малко размера на по-голям обект, като внимателно прецените и промените типовете на целочислените му полета.
- Може да използвате CLR Profiler за да определите размера на референтен тип.

Отражение на типовете

Избягвайте използването на отражение на типовете (reflection), когато е възможно. Ако се питате каква е цената на reflection, тя е такава, че не можете да си я позволите. Ето защо и класът **ObjectPool**, даден като пример в една от следващите точки, не използва reflection за да създава обекти. Отражението на типовете е полезно и мощно средство, но сравнено код, преминал през JIT компилатора е много пъти по-бавно.

Премахнете създаването на временни обекти, които могат да бъдат избегнати с цената на малко повече код

Например, ако трябва да сортирате CSV файл (файл, в който данните в редовете са разделени със запетаи) и първата колона съдържа ключа за сортиране, може да напишете следния клас за сравнение на редовете:

```
sealed class SlowComparer : IComparer
{
   private readonly char mDelimiter;

   public SlowComparer(char aKeyDelimiter)
   {
      mDelimiter = aKeyDelimiter;
   }

   public int Compare(object aObj1, object aObj2)
   {
      string key1 = (aObj1 as string).Split(mDelimiter)[0];
      string key2 = (aObj2 as string).Split(mDelimiter)[0];
      int len = Math.Min(key1.Length, key2.Length);
      return String.Compare(key1, 0, key2, 0, len);
   }
}
```

Методът за сравнение **Compare(…)**, показан в горната фигура първо разделя реда на колони, използвайки метода **Split(…)** на класа **System.String**, а после ползва първата колона като ключ. Извикването на **Split(…)** създава масив състоящ се от низове, като както масива, така и низовете са заделени в динамичната памет.

С малко повече усилия, при положение, че знаете, че редовете са разделени със запетаи, може да извлечете ключовете за сортиране, без да създавате ненужен разход на памет:

```
sealed class FastComparer : IComparer
{
   private readonly char mDelimiter;

   public FastComparer(char aKeyDelimiter)
   {
```

```
      mDelimiter = aKeyDelimiter;
   }

   public int Compare(object aObj1, object aObj2)
   {
      string str1 = aObj1 as string;
      string str2 = aObj2 as string;
      int pos1 = str1.IndexOf(mDelimiter, 0);
      int pos2 = str2.IndexOf(mDelimiter, 0);
      int len = Math.Min(pos1, pos2) + 1;
      return String.Compare(str1, 0, str2, 0, len);
   }
}
```

Сортирането на един и същ неподреден масив, състоящ се от 100 000 низа от по 100 символа, с ключ между 5 и 10 символа с **`FastComparer`** е повече от 20 пъти по-бързо от това с помощта на **`SlowComparer`**.

Минимизирайте броя на записите на указатели към вашите обекти, особено онези, които се правят в по-стари обекти

Малко по-горе обяснихме как кешът на процесора не трябва да се пренебрегва. Когато в поле, намиращо се в по-стар обект (такъв, който се е преместил в по-горно поколение) запишете референция вие първо предизвиквате GC да обнови таблицата с референции, неподлежащи на GC и второ, но не по-малко важно, "докосвате" стар обект, който е много вероятно да е излязъл от кеша на процесора. Внимателно прегледайте дизайна си, тъй като би трябвало по-често да ви се налага да записвате референция към стар обект в поле на нов, отколкото обратно.

Използвайте възможно най-малко финализатори

Ако е нужно разбийте обектите си на подобекти, за да го постигнете – това също важи за разделянето на топли и студени обекти.

Запознайте се с инструмента CLR Profiler

Преглеждайте кода, имплементиращ често използвани структури от данни и го оптимизирайте с CLR Profiler за да сте сигурни, че употребявате паметта ефективно и за да работи GC най-добре за вас.

CLR Profiler (бившият Allocation Profiler) е полезна програма, написана от екипа на Microsoft, която използва програмните интерфейси за профилиране на CLR код (CLR profiling APIs), събирайки и визуализирайки по подходящ начин информация за събития като:

- извикване на метод
- връщане от извикан метод
- заделяне на памет за обект
- почистване на паметта и др.

След като необходимата информация от събитията е събрана, можете да използвате CLR Profiler за да разгледате заделянето на памет и поведението на GC за вашето приложение, включително взаимодействието между йерархичното извикване на методите ви и шаблоните, по които заделяте памет.

Изучаването на CLR Profiler си струва, защото за много приложения, имащи проблеми с производителността, разбирането на шаблона на заделянето на памет за вашите данни помага за намаляването на working set паметта и за създаването на бързи компоненти и приложения.

CLR Profiler (с включена документация) може да се свали свободно от: http://www.microsoft.com/downloads/details.aspx?FamilyId=86CE6052-D7F4-4AEB-9B7A-94635BEEBDDA&displaylang=en

Проектирайте, мислейки за ефективността

За някои проекти производителността е с малко или без значение, а за други тя е най-важната характеристика на продукта. Преждевременната оптимизация е корена на всяко зло, но и неханието по отношение на ефективността води до много проблеми. Вие сте професионалисти, ето защо трябва да знаете цената на нещата. Ако не я знаете, пък и дори да си мислите, че я знаете – премервайте често.

Техниката "пулинг на ресурси"

Пулинг на ресурси (resource pooling) е програмна техника за подобряване на производителността при работа с ресурси, които се създават или унищожават "скъпо" (бавно).

Тази техника се използва от много рамки на приложения (frameworks), включително от COM+ за да не създава и унищожава непрекъснато обекти, а в .NET Framework – при управлението на най-разнообразни, най-вече неуправляеми ресурси като:

- връзки към бази данни (connection pooling);
- нишки (thread pooling);
- и др.

Пулът обикновено представлява списък от обекти, които се създават предварително (например при инициализация на приложението), а после се "раздават" при поискване. Клиентите взимат обекти от пула, използват ги известно време и след като вече не им трябват, не ги унищожават, а ги връщат обратно в пула.

Дизайнът на пула може да бъде различен, според нуждите на вашето приложение:

- типизиран (само за определен тип обекти) или не;
- изискващ имплементация на определен интерфейс от обектите или не;

- позволяващ или не инстанциране на пула, т.е. Сек[17] (Singleton) както **`System.ThreadPool`** или обикновен клас;
- позволяващ или не задаване на броя на предварително създадените обекти в пула;
- с ограничен брой елементи (като **`System.ThreadPool`**) или безкраен, като при свършване на елементите в пула се създават и добавят нови;
- безопасен за работа в многонишково приложение (thread-safe) или такъв, който трябва да се синхронизира ръчно от потребителя;
- разширяем (който може да се наследява) или не (sealed class);
- създаващ обектите вътрешно (ако е типизиран) или посредством стратегия за тяхното създаване, например делегат (delegate).

Примерна имплементация на пул от ресурси

Ще ви дадем пример за разширяем, нетипизиран, thread-safe пул от обекти, чието създаване се осъществява или посредством обект-стратегия[18] или посредством прототипна инстанция.

Ако пулът е инициализиран с прототип, се създава стратегия по подразбиране и пулът се инициализира с нея. Обектът-прототип трябва да имплементира следния интерфейс:

```
public interface IobjectPrototype
{
   // Обектът връща пълно копие на себе си
   IobjectPrototype DeepClone();
}
```

Обектът-стратегия трябва да имплементира интерфейс, позволяващ динамичното създаване и унищожаване на обекти за целите на пула:

```
public interface IobjectPoolStrategy
{
   // Създава нов обект
   object Create();

   // Унищожава обект, създаден от Create
   void Destroy(object aInstance);
}
```

[17] виж "Шаблони за дизайн", шаблон "Сек"

[18] виж "Шаблони за дизайн", шаблон "Стратегия"

Както споменахме, ако пулът е инициализиран с прототип, ще използва стратегията по подразбиране за създаване на обекти и елементарна имплементация на унищожаването им:

```
// Потребителите на пула могат да дефинират клас,
// който наследява DefaultObjectPoolStrategy за да
// не имплементират целия интерфейс
public class DefaultObjectPoolStrategy : IobjectPoolStrategy
{
   // Ако наследниците не използват прототип, просто подават null
   public DefaultObjectPoolStrategy(IobjectPrototype aPrototype)
   {
      mPrototype = aPrototype;
   }

   // Ако наследниците не предефинират метода,
   // се използва прототипа
   public virtual object Create()
   {
      if (mPrototype == null)
      {
         // Едновременното непредефиниране на метода
         // и неподаването на прототип е грешка
         string message = String.Format(
            "{0} instantiated without prototype",
            GetType().Name);
         throw new InvalidOperationException(message);
      }

      return mPrototype.DeepClone();
   }

   // Имплементация по подразбиране на Destroy метода
   public virtual void Destroy(object instance)
   {
      if (instance != null)
      {
         // Ако обектът имплементира Idisposable го унищожаваме
         Idisposable disp = instance as Idisposable;
         if (disp != null)
         {
            disp.Dispose();
         }
      }
   }

   private IobjectPrototype mPrototype;
}
```

Ето и имплементация на пула. Тя съдържа подробни коментари и затова няма да я обсъждаме по-нататък.

```
// След инстанциране, с пула се работи чрез:
//   метода Draw() за извличане на обект от пула, например:
//     object obj = pool.Draw();
//   метода Return() за връщане на обект в пула, например:
//     pool.Return(obj);
//   свойството IsLimited, което връща дали пулът е ограничен
// Наследяваме ResourceWrapperBase за да извикаме Destroy()
// за всички обекти в пула, когато той се унищожава
public class ObjectPool : ResourceWrapperBase
{
   private const int DEFAULT_MIN_OBJECTS = 4;
   private const int UNLIMITED = -1;

   // обект-стратегия за създаване и унищожаване на обекти
   private IobjectPoolStrategy mStrategy;

   private Stack mItems; // хранилище за обектите в пула
   private int mInitialObjects; // първоначален брой обекти
   private int mMaxObjects; // максимален брой обекти

   // Създава обекти чрез прототип. Пулът е неограничен, а
   // първоначалният брой е по подразбиране
   public ObjectPool(IobjectPrototype aPrototype) :
      this(aPrototype, DEFAULT_MIN_OBJECTS)
   {
   }

   // Създава aInitialObjects обекта предварително
   // чрез прототип. Пулът е неограничен
   public ObjectPool(IobjectPrototype aPrototype,
      int aInitialObjects) :
      this(aPrototype, aInitialObjects, UNLIMITED)
   {
   }

   // Създава aInitialObjects обекта чрез прототип.
   // Пулът е ограничен до aMaxObjects
   public ObjectPool(IobjectPrototype aPrototype,
      int aInitialObjects, int aMaxObjects)
   {
      if (aPrototype == null)
      {
         throw new ArgumentNullException("prototype");
      }

      // Действителната инициализация се случва в Init()
      Init(aInitialObjects, aMaxObjects, aPrototype, null);
   }

   // Създава обекти чрез стратегия. Пулът е неограничен.
```

```
// Първоначалният брой е по подразбиране
public ObjectPool(IobjectPoolStrategy aStrategy) :
   this(aStrategy, DEFAULT_MIN_OBJECTS)
{
}

// Създава aInitialObjects обекта чрез стратегия.
// Пулът е неограничен
public ObjectPool(IobjectPoolStrategy aStrategy,
   int aInitialObjects) :
   this(aStrategy, aInitialObjects, UNLIMITED)
{
}

// Създава aInitialObjects обекти чрез стратегия.
// Пулът е ограничен до aMaxObjects
public ObjectPool(IobjectPoolStrategy aStrategy,
   int aInitialObjects, int aMaxObjects)
{
   if (aStrategy == null)
   {
      throw new ArgumentNullException("strategy");
   }

   Init(aInitialObjects, aMaxObjects, null, aStrategy);
}

// Метод за инициализация. Извиква се от конструкторите
private void Init(int aInitialObjects,
   int aMaxObjects, IobjectPrototype aPrototype,
   IobjectPoolStrategy aStrategy)
{
   // Ако aMaxObjects == UNLIMITED, пулът е неограничен
   if (aInitialObjects < 0 ||
      (aMaxObjects != UNLIMITED &&
       aMaxObjects < aInitialObjects))
   {
      throw new ArgumentException(
        "initialObjects < 0 or maxObjects < initialObjects");
   }

   mInitialObjects = aInitialObjects;
   mMaxObjects = aMaxObjects;
   mStrategy = aStrategy;
   if (mStrategy == null)
   {
      mStrategy = new DefaultObjectPoolStrategy(aPrototype);
   }

   // Създаваме минимума обекти
```

```
    mItems = new Stack(mMaxObjects);
    for (int i = 0; i < mInitialObjects; ++i)
    {
      mItems.Push(CreateObject());
    }
  }

  // Връща дали пулът е ограничен
  public bool IsLimited
  {
    get
    {
      return mMaxObjects != UNLIMITED;
    }
  }

  // Извлича свободна инстанция от пула. Ако пулът е
  // празен, се разширява, а ако не може да се разшири,
  // се получава изключение
  public object Draw()
  {
    lock (mItems)
    {
      // Тук сме сигурни, че се изпълнява само една нишка
      if (mItems.Count > 0)
      {
        return mItems.Pop();
      }

      // Тук сме попаднали, ако първоначално или
      // след заключването е нямало свободни елементи.
      // Проверяваме може ли да разширим пула
      int itemsToAdd;
      if (mMaxObjects == UNLIMITED)
      {
        itemsToAdd = 1;
      }
      else
      {
        itemsToAdd = mMaxObjects - mInitialObjects;
        if (itemsToAdd == 0)
        {
          throw new InvalidOperationException(
            "The pull is empty and can not grow further.");
        }
      }

      for (int i = 0; i < itemsToAdd; ++i)
      {
        mItems.Push(CreateObject());
```

```
        }

        // Гарантирано е, че имаме поне един елемент в пула
        return mItems.Pop();
      }
    }

    // Връща дадена инстанция обратно в пула
    public void Return(object aInstance)
    {
      if (aInstance == null)
      {
        throw new ArgumentNullException("instance");
      }

      lock (mItems)
      {
        mItems.Push(aInstance);
      }
    }

    // Наследниците на пула могат да предефинират
    // създаването и унищожаването на обекти
    protected virtual object CreateObject()
    {
      return mStrategy.Create();
    }

    protected virtual void DestroyObject(object aInstance)
    {
      mStrategy.Destroy(aInstance);
    }

    // Метод от ResourceWrapperBase – унищожава обектите в пула
    protected override void DisposeManagedResources()
    {
      foreach (object instance in mItems)
      {
        DestroyObject(instance);
      }
      base.DisposeManagedResources();
    }
}
```

Класът **ObjectPool** може да бъде подобрен, особено по отношение на работата му в многонишкова конкурентна среда:

- Можете да добавите събитие (**AutoResetEvent**), което се сигнализира при наличието на обект в пула и да промените кода на **Draw**, така че

да разчита на това събитие и при празен пул да чака изчаква докато някой върне обект в пула.

- Можете да добавите метод **`object TryDraw(int milliseconds)`**, който да се опитва да получи обект от пула в рамките на интервала, посочен в **`milliseconds`**, след което да изхвърли изключение или да върне **`null`**.

Всичко това оставяме като ужасяващо упражнение за читателя.

Упражнения

1. Какво знаете за автоматичното управление на паметта и ресурсите в .NET Framework? Какви са предимствата и недостатъците на автоматичното управление на паметта? Как работи т. нар. garbage collector?
2. Какво знаете за финализацията и интерфейса **`IDisposable`** в .NET Framework? Кога се използват? Как се реализират?
3. С помощта на класа **`ResourceWrapperBase`** реализирайте обвивка на неуправлявания ресурс "Windows шрифт".
4. Напишете клас **`BufferedConsole`**, който предоставя буфериран изход към конзолата чрез метода си **`Write(string)`**. Класът трябва да съдържа в себе си буфер с размер 50 байта, в който се добавят изпратените низове. При препълване на буфера данните от него трябва да се отпечатват на конзолата. Имплементирайте финализация и **`IDisposable`** и при почистване на ресурсите отпечатвайте буфера на конзолата.
5. Реализирайте примерна програма, която използва класа **`BufferedConsole`** за да печата различни съобщения в конзолата. Използвайте конструкцията **`using`** в C# за да освободите правилно инстанцията на класа **`BufferedConsole`**.
6. Реализирайте правилно освобождаване на инстанцията на **`BufferedConsole`** от предходната задача без да използвате конструкцията **`using`**, а чрез **`try...finally`** конструкция.
7. Реализирайте примерна програма, която печата по конзолата чрез класа **`BufferedConsole`** и разчита на финализацията за да не се губят данните от буфера при почистване на паметта. Защо този подход трябва да се избягва пред възможността ресурсите да се почистят ръчно?
8. Реализирайте метод, който по дадени цели числа N и K връща броя на комбинациите без повторение от N елемента, K-ти клас. Използвайте за изчисленията триъгълника на Паскал и слаби референции, в които съхранявайте отделните му редове.
9. Реализирайте прост пул от обекти от тип **`Resource`**. Пулът трябва да не е защитен от конкурентен достъп (thread unsafe), да няма ограничение

за броя създадени едновременно обекти, да не създава предварително никакви обекти и да съхранява освободените инстанции в стек.

Използвана литература

1. Светлин Наков, Управление на паметта и ресурсите – http://www.nakov.com/dotnet/lectures/Lecture-10-Memory-Management-v1.0.ppt
2. Георги Иванов, Управление на паметта и ресурсите – http://www.nakov.com/dotnet/2003/lectures/Memory-management-finalization.doc
3. Jeffrey Richter, Applied Microsoft .NET Framework Programming, Microsoft Press, 2002, ISBN 0735614229
4. Tushar Agrawal, Memory Management in .NET – http://www.c-sharpcorner.com/Code/2003/Nov/MemoryManagementInNet.asp
5. MSDN Training, Programming with the Microsoft® .NET Framework (MOC 2349B), Module 9: Memory and Resource Management
6. MSDN Library – http://msdn.microsoft.com
7. MSDN Magazine, Jeffrey Richter, Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework – http://msdn.microsoft.com/msdnmag/issues/1100/GCI/default.aspx
8. MSDN Magazine, Jeffrey Richter, Garbage Collection – Part 2: Automatic Memory Management in the Microsoft .NET Framework – http://msdn.microsoft.com/msdnmag/issues/1200/GCI2/

Глава 25. Асемблита и разпространение

Автор

Галин Илиев

Необходими знания

- Базови познания за .NET Framework и CLR (Common Language Runtime)
- Базови познания за общата система от типове в .NET (Common Type System – CTS)
- Познания по езика C#
- Познаване на инструментите от .NET Framework SDK
- Базови познания за Windows Installer

Съдържание

- Какво е асембли? Манифест на асембли
- Конфигурационни файлове
- Как CLR намира асемблитата?
- Global Assembly Cache
- Разпространение и инсталиране на програмни пакети
- Инсталационни компоненти
- COM базирани обекти
- Сървърни компоненти (Serviced Components)
- Настройки на Internet Information Server (IIS)
- Промяна на регистрите на Windows
- Споделени инсталационни компоненти (Merge Modules)
- CAB файлове
- Локализиране
- Debug Symbols
- Инсталационни стратегии
- Създаване на MSI инсталационни пакети с VS.NET

В тази тема...

В настоящата тема ще разгледаме най-малката съставна част на .NET приложенията – асемблитата. Ще разгледаме за какво служат, от какво се състоят и как могат да се конфигурират. Ще се спрем на различните техники за разпространение на готовия софтуерен продукт на клиентските работни станции и на някои избрани техники за създаване на инсталационни пакети.

Асемблитата в .NET Framework

Асемблитата са основна съставна част на всеки софтуерен продукт, базиран на .NET Framework. Те са най-малката и основна част при разпространение на .NET приложения. Асемблитата се състоят от компилирани .NET типове (интерфейси, класове, структури и др.), метаданни и ресурси (**.bmp**, **.jpeg**, **.ico** файлове, **.resource** и **.resx** ресурси и други). Компилираните типове представляват изпълним програмен код във вид на инструкции на междинния език IL. Метаданните описват асемблитата и типовете в тях. Ресурсите могат да бъдат вградени или записани като външни файлове.

Асемблитата могат да бъдат статични и динамични. Статичните асемблита се съхраняват във файл в portable executable (PE) формат, докато динамичните се изпълняват директно от паметта и не се записват (във файл) преди изпълнението им .NET Framework предлага стандартни средства и инструменти за създаване на динамични асемблита и позволява тяхното изпълнение и съхранение с помощта на класовете от пространството **System.Reflection.Emit**.

Асемблитата съдържат IL код за изпълнение

Асемблитата съдържат компилирани .NET типове – програмен код във вид на инструкции на езика Intermediate Language (IL), който се изпълнява от CLR чрез компилация до машиннозависим код. Важно условие за изпълнение на IL кода е наличието на метаданни за асемблито и асембли манифест.

Асемблитата се записват във файлове, които са във формат PE (portable executable). Тези файлове най-често носят разширения **.exe** или **.dll**. Всеки преносим изпълним файл (PE файл) може да има входна точка за изпълнение – функция **DllMain(…)**, **WinMain(…)** или **Main(…)**, съответно за динамични библиотеки, Windows GUI приложения и конзолни приложения. Входната точка може да е най-много една.

Асемблитата формират граница за сигурността (security boundary)

Кодът, който се съдържа в дадено асембли, изисква определени права за достъп и изпълнение. Асемблитата са единица, която може да изисква и получава определени права (permissions). Когато се създава асембли, неговият разработчик може да посочи минимален набор от права, които асемблито задължително изисква, за да работи.

Дали определени права ще се дадат на дадено асембли зависи от политиките за сигурност на .NET Framework и т. нар. доказателства, които има асемблито – цифров подпис, силно име (strong name), местоположение (URL, UNC) и др. Например ако дадено асембли се зареди от

интернет адрес, който не е указан като сигурен (trusted) в Internet Explorer, то асемблито се стартира с ограничени права, част от които е ограничен достъп до файловата система, като файлове се записват в т. нар. Isolated Storage.

Асемблитата формират граница за типовете (type boundary)

Всяко асембли обгръща типовете, които съдържа. Идентичността на типовете е свързана с името на асемблито, в което се намират. Това означава, че типът **`MyType`**, деклариран в **`assembly1.dll`**, не е еднакъв с типа **`MyType`**, деклариран в **`assembly2.dll`**. Това позволява по-голяма гъвкавост и независимост на имената при капсулиране на функционалност в асемблитата.

Асемблитата формират граница на видимостта (reference scope boundary)

Манифестът на асемблитата (ще се спрем на него малко по-нататък в настоящата тема) съдържа метаданни, които се използват за намиране на типовете и ресурсите, включени в асемблито. Манифестът определя типовете и ресурсите, които са видими извън границите на асемблито (от други асемблита). В него се описват също и асемблитата, които се изискват, за да се изпълни основното асембли.

Асемблитата формират граница на версиите (version boundary)

Асемблито е най-малката единица, която притежава версия в CLR. Версия се задава на всички типове и ресурси в дадено асембли заедно като един обект.

В .NET Framework е възможно различни версии на едно и също асембли да съществуват и да се изпълняват едновременно, без да си пречат. Това решава много проблеми, предизвикани от конфликти във версиите.

Манифестът описва точно версията на асемблито, както и версиите на асемблитата, които се изискват, за да се изпълни то.

Асемблитата са единица за споделяне

Асемблитата са най-малката единица, която може да бъде споделена между няколко .NET приложения. Основен начин на споделяне е да се даде на асемблито силно име (виж. Силно именуване на асембли) и да се постави в GAC (Global Assembly Cache). Друг начин е да се инсталира като частно асембли към дадено приложение. Подробно ще разгледаме тези техники в частта "Разпространение на асемблита".

Асемблитата са единици за разпространение (deployment units)

Асемблитата формират основна програмна единица за разпространение. Когато се стартира едно .NET приложение са му необходими само асемблитата, които се извикват първоначално. Останалите асемблита (като ресурси за локализация или допълнителни модули) се зареждат при първото им поискване (on demand). Това позволява приложенията да се поддържат малки и удобни при първоначално разпространение. Тази възможност е особено важна при технологията **.NET Zero Deployment**, която ще опишем по-късно в настоящата тема.

Метаданни и манифест на асембли

Всяко асембли, независимо дали е статично или динамично, съдържа в себе си метаданни (информация, която го описва).

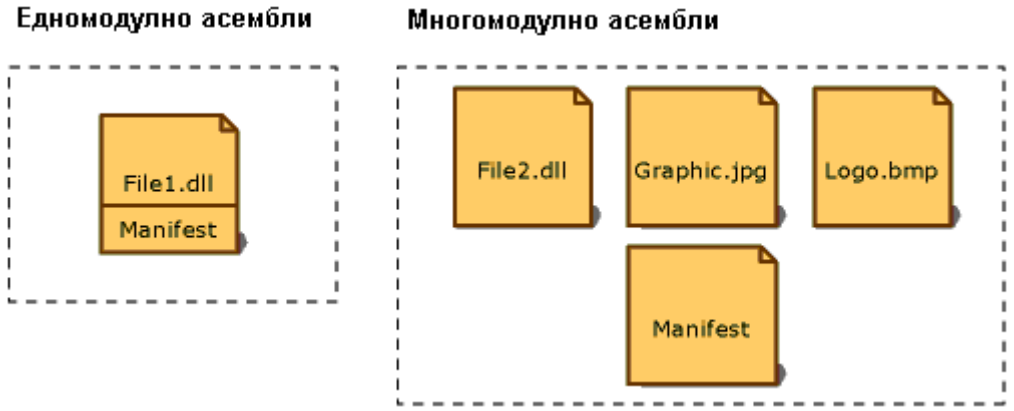
Метаданните включват описание на съдържаните в асемблито типове и информация за него самото.

Манифест на асембли

Информацията за асемблито описва как са свързани съдържаните елементи помежду си – това е т. нар. **манифест**. Манифестът съдържа всички метаданни, нужни за описанието на идентичността на асемблито, информация за неговата версия, необходимите му права, асемблитата и версиите им, нужни за изпълнението му, както и допълнителна информация, необходима за извличането на типовете и ресурсите.

Манифестът може да се съдържа в самото асембли (в неговия `.exe` или `.dll` преносим изпълним файл) заедно с останалите ресурси или като самостоятелен файл, който съдържа само информацията на манифеста.

Следващата илюстрация показва различните начини, по които се съхранява манифеста в асемблитата:



При асемблита, които съдържат един файл, манифестът е вмъкнат в PE файла и образува асембли от един файл. Възможно е създаването на многомодулно асембли с външен манифест или манифестът може да е вмъкнат в един от файловете.

Съдържание на манифеста

Следната таблица показва съдържанието на манифеста:

Информация		Описание
Идентификация	**Име**	**Текст, указващ име на асембли.**
	Версия	**Съдържа 4 идентификатора разделени един от друг с точка във формат:** `Major.Minor.Build.Revision` **(напр.** `3.44.1234.5543`**).**
	Култура	**RFC1766 низ, указващ регионални настройки (locale) на асемблито – например "**`en-US`**". Тази информация трябва да се използва, когато се създава сателитно асембли[19], съдържащо специфична културна и езикова информация. Възможно е асемблито да е с неутрална култура. (Асемблита с информация за културата автоматично се възприемат като сателитни).**
	Информация за силно именуване	**Съдържа публичния ключ, използван за цифровия подпис на асемблито, ако то притежава силно име.**
Списък на файловете, включени в асемблито		**Хеш стойност и име на файл, включен в асемблито. Забележка: Всички файлове трябва да са в папката, където се намира е файлът, съдържащ манифеста.**
Съдържани типове		**Тази информация се използва по време на изпълнение за зареждане на типовете декларирани в асемблито.**
Външни асемблита		**Списък с всички външни асемблита, които са статично свързани. Всяко свързано асембли е описано с име и метаданни (версия, култура, и т.н.) и публичен ключ, ако е силно именувано.**
Изисквани права		**Необходими права, за да се изпълни асем-**

[19] Сателитното асембли представлява асембли, което съдържа само ресурси без изпълним код [7].

за достъп	блито.

Първите четири елемента (име, версия, култура, публичен ключ) образуват **идентификацията** на асембли – неговото силно име.

Атрибути за работа с манифест

В .NET Framework има няколко атрибута, чрез които можем да влияем на компилацията на асемблитата и да променяме метаданните в техния манифест. Тези атрибути се записват в сорс кода на асемблито и при компилация модифицират метаданните. Ето по-важните от тях:

Атрибут на манифеста	Описание
`AssemblyCultureAttribute`	**RFC1766 низ указващ към коя култура принадлежат ресурсите, съдържани от асемблито. Формата е** `"език"-"страна или регион"` **– например** `"en-US"` **за US English или** `"bg-BG"` **за ресурси на български език. Културата може да бъде също неутрална, което показва, че асемблито съдържа ресурси за културата по подразбиране.** **Забележка: CLR третира всяко асембли с този атрибут за сателитно.**
`AssemblyFlagsAttribute`	**Указва дали асемблито поддържа едновременно различни версии в един компютър, в един процес или в един домейн на приложение (т. нар. side-by-side изпълнение, което ще разгледаме по-подробно в секцията за Global Assembly Cache).**
`AssemblyVersionAttribute`	**Съдържа 4 идентификатора разделени с точка:** `Major.Minor.Build.Revision` **(напр.** `3.44.1234.5543`**).**
`AssemblyCompanyAttribute`	**Текстов низ, указващ име на компанията, производител на асемблито.**
`AssemblyCopyrightAttribute`	**Текстов низ, описващ авторски права.**

AssemblyFileVersionAttribute	**Текстов низ, указващ Win32 файлова версия. По подразбиране това е версията на асемблито.**
AssemblyInformational-VersionAttribute	**Текстов низ, указващ версия, която не се използва от CLR. Това може да бъде версия на продукт или друга, която носи информация за разработчиците.**
AssemblyProductAttribute	**Текстов низ, указващ име на софтуерен продукт.**
AssemblyTrademarkAttribute	**Текстов низ, указващ търговска марка.**
AssemblyConfigurationAttribute	**Текстов низ, указващ конфигурация на асемблито (Release или Debug). CLR не използва този атрибут.**
AssemblyDefaultAliasAttribute	**Текстов низ, указващ псевдоним по подразбиране, който ще се използва при свързване от външни асемблита. Може да бъде лесно за използване име (докато името на асемблито може да не е). Може да е също съкратена форма на пълното име на асемблито.**
AssemblyDescriptionAttribute	**Текстов низ, съдържащ кратко описание на съдържанието и целта на асемблито.**
AssemblyTitleAttribute	**Текстов низ, указващ лесно име на асемблито (напр. Microsoft Common Dialog Control).**

Атрибути за работа с манифест – пример

Ето един пример, в който чрез атрибути се задават метаданни за асемблито:

AssemblyInfo.cs

```
using System.Reflection;

[assembly: AssemblyTitle("Advanced Toolbar Control")]
[assembly: AssemblyDescription(
```

```
  "Advanced Toolbar Windows Forms Control")]
[assembly: AssemblyConfiguration("Release")]
[assembly: AssemblyCompany("Software Abuse Corp.")]
[assembly: AssemblyProduct(
  "Software Abuse Windows Controls Library")]
[assembly: AssemblyCopyright(
  "(c) 2005 by Software Abuse Corp.")]
[assembly: AssemblyCulture("")]
[assembly: AssemblyVersion("3.22.*")]
```

Във VS.NET метаданните за асемблито обикновено се записват във файла **AssemblyInfo.cs**, който се създава автоматично с всеки нов проект (освен ако проектът не е от тип "Empty Project"). В примера сме използвали този файл, за да дефинираме метаданни за асемблито, до което се компилира текущия проект – контролата "Advanced Toolbar Control".

Създаване на многомодулно асембли

След като обяснихме понятието "асембли", нека сега навлезем малко по-дълбоко. Асемблито може да състои от един файл (или модул – това е най-често срещаната форма) или от няколко такива. Асемблита, които се състоят от повече то един файл се наричат "многомодулни асемблита".

Причините да се използват многомомодулни асемблита са [8]:

1. Комбиниране на модули, написани на различни езици в едно асембли. Всеки модул се компилира отделно с подходящ компилатор и се свързват чрез Assembly Linker **(Al.exe)**.
2. При използване на No-Touch Deployment (.NET Zero Deployment) се сваля само изисквания модул, което намалява мрежовия трафик и времето за стартиране на приложението. Ще разгледаме по подробно тази технология по-надолу.
3. При използване на файл с политика на издателя (publisher policy file) – виж Създаване на Publisher Policy File – конфигурационния файл се приема за първи модул от асемблито с политиките на издателя.

Да разгледаме стъпките, чрез които можем да създадем асембли, което се състои от няколко файла.

Тъй като VS.NET не може да създава многомодулни асемблита, ако се наложи да използваме такива, можем да ги създадем от командния ред чрез подходящи извиквания на C# компилатора (**csc.exe**).

Стъпка 1: Компилиране на модул, който ще бъде използван от други асемблита

Създаваме файл **Utilities.cs** със следното съдържание:

```
Utilities.cs
```

```
// Assembly building in the .NET Framework - example
using System;

namespace Utilities
{
   public class ConsoleFunctions
   {
      public void ConsoleWrite()
      {
         System.Console.WriteLine("A line from Utilities!");
      }
   }
}
```

Компилираме го с командата:

```
csc /target:module Utilities.cs
```

След успешното компилиране е създаден файл **Utilities.netmodule** в същата папка.

Стъпка 2: Компилиране на асембли, което използва вече създадения модул

Създаваме файл **Client.cs** в същата папка:

Client.cs

```
using System;
using Utilities; // The namespace created in Utilities.netmodule

class MainClientApp
{
   // Static method Main is the entry point method.
   public static void Main()
   {
      ConsoleFunctions myFunctions = new ConsoleFunctions();
      Console.WriteLine("Client code executes");

      // Call function from module Utilities.netmodule
      myFunctions.ConsoleWrite();
   }
}
```

Можем да го компилираме с командата като **/addmodule** указва на компилатора, че ще се използва методи от модула **Utilities.netmodule**:

```
csc /target:module Client.cs /addmodule:Utilities.netmodule
```

Този път компилаторът създава файла **Client.netmodule**.

Стъпка 3: Свързване на модулите в асембли чрез Assembly Linker

В тази стъпка ще използваме инструмента Assembly Linker **(Al.exe)**, за да свържем модулите в едно асембли. Стартираме командата **al.exe** със следните параметри (в един ред):

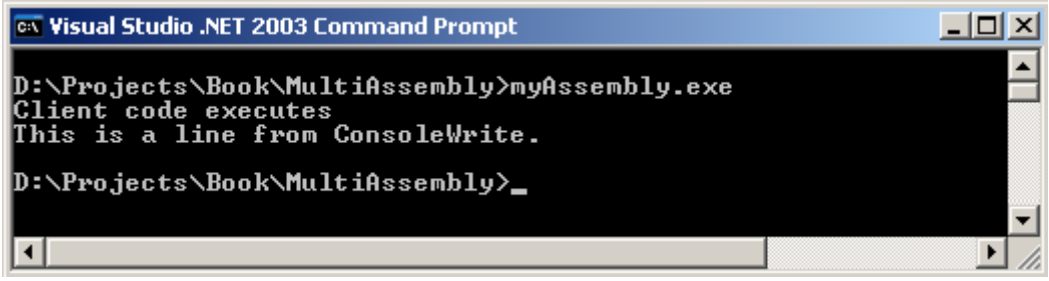
```
al  Client.netmodule Utilities.netmodule
/main:MainClientApp.Main /out:myAssembly.exe /target:exe
```

Резултатът от изпълнението на тази команда е файлът **myAssembly.exe**.

Параметрите, които подадохме, имат следните значения:

- Първите два параметъра са модулите, които създадохме в предишните стъпки.
- **/main** – указва входната точка на създаваното асембли (метода, който ще получи управлението при стартиране).
- **/out** – указва името на файла, който ще бъде създаден.
- **/target:exe** – указва да се създаде изпълним файл за конзола.

Нека видим резултата от изпълнението от създаденото асембли:



За да се зареди асемблито трябва да са налични всички модули. Това се налага от факта, че асемблито се компилира от IL код до машиннозависим код от JIT компилатора.

Разглеждане на манифеста на асембли с ildasm

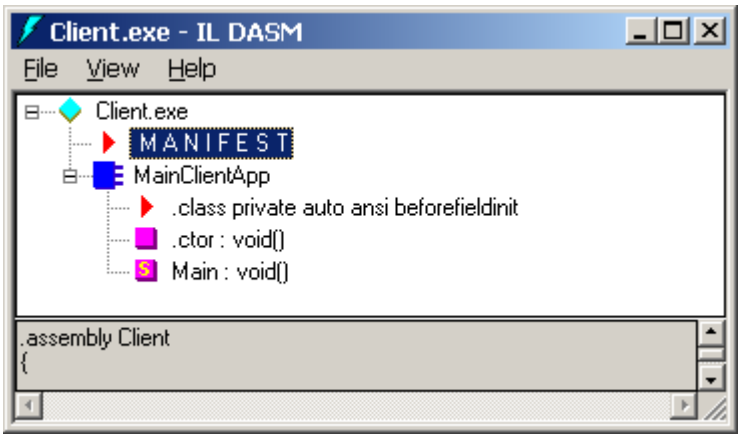
Вече се запознахме какво представляват манифестите на асемблитата и как можем да променяме метаданните, които се записват в тях при компилация. Нека сега видим как можем да разгледаме манифеста на съществуващо асембли.

MSIL Disassembler (**ildasm.exe**) е придружаващ инструмент към MSIL Assembler (**ilasm.exe**). С този инструмент може да се разглежда съдържанието на асемблитата и модулите. MSIL Disassembler е част от .NET

Framework SDK и се намира по подразбиране в директорията **%ProgramFiles%\Microsoft Visual Studio .NET 2003\SDK\v1.1\bin**.

За да разгледаме метаданните и манифеста на асемблито **Client.exe**, получено в горния пример, ще го отворим с MSIL Disassembler чрез следния команден ред, който е изпълнен във Visual Studio .NET 2003 Command Prompt (стартира се от бутона Start->Programs->Microsoft Visual Studio .NET 2003->Visual Studio .NET Tools->Visual Studio .NET 2003 Command Prompt):

```
ildasm Client.exe
```



Двойно щракване с левия бутон на мишката върху "**M A N I F E S T**" ще отвори прозорец със съдържанието на манифеста:

MANIFEST

```
.module extern Utilities.netmodule
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .hash = (39 E4 FC 93 06 1A 5E 87 28 97 0F 5F AC AD B4 21   /
           EF 71 E8 D2 )                                     /
  .ver 1:0:5000:0
}
.assembly Client
{
  // --- The following custom attribute is added automatically
  //  .custom instance void [mscorlib]System.Diagnostics.Debug
  //
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.file Utilities.netmodule
    .hash = (99 F6 4E ED B7 ED 77 FC 5E 43 4F E7 F8 C2 05 B4
             57 8F 4B 3B )
.class extern public Utilities.ConsoleFunctions
{
  .file Utilities.netmodule
  .class 0x02000002
}
.module Client.exe
// MVID: {45381C31-9D3F-4905-9D8E-0F2F26730D79}
.imagebase 0x00400000
.subsystem 0x00000003
```

Както се вижда, манифестът дефинира асемблито **`Client`**, което съдържа модула **`Client.exe`** и реферира външния модул **`Utilities.netmodule`** от файла **`Utilities.netmodule`**, както и външното асембли **`mscorlib`** и външния клас **`Utilities.ConsoleFunctions`**.

Повече информация относно MSIL Disassembler можете да намерите в MSDN в статията http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpconMSILDisassemblerIldasmexe.asp.

Силно именуване на асемблии

С цел повишаване на сигурността и намаляване на конфликтите с версиите в .NET Framework са въведени т. нар. силно именувани асемблита. **Силното име** на асембли се образува от неговите име, версия, култура и публичен ключ. Всяко силно именувано асембли е цифрово подписано с частния ключ, съответстващ на публичния ключ, който участва в силното му име. Пример за силно име на асембли е следният низ:

```
CompanyNamespace.Controls.Design.v3.2,Version=3.2.111251.37,
Culture=neutral, PublicKeyToken=8ty5c3176f5cd04e
```

Силно именуваните асемблита имат някои предимства пред обикновените асемблита:

- Гарантира се уникалността, с помощта на двойка ключове (public / private key pair). Никой не може да създаде асембли със същото име, без да притежава частния ключ. Асембли, създадено с един частен ключ, се различава от асембли, създадено с друг частен ключ.

Защитава родословието на версиите. Нова версия, която е създадена без притежаването на същия частен ключ като предходната, лесно може да бъде идентифицирана.

- Защитава се целостта на асемблито и дадените му права (permissions). По този начин се гарантира, че подмяна на асембли, с цел използване на неговите права (permissions) е невъзможно.

Само силно именувани асемблита може да се добавят в областта Global Assembly Cache, която ще дискутираме след малко.

Създаване на силно именувано асембли – пример

В следващия пример ще създадем силно именувано асембли и ще покажем особеностите в неговия манифест.

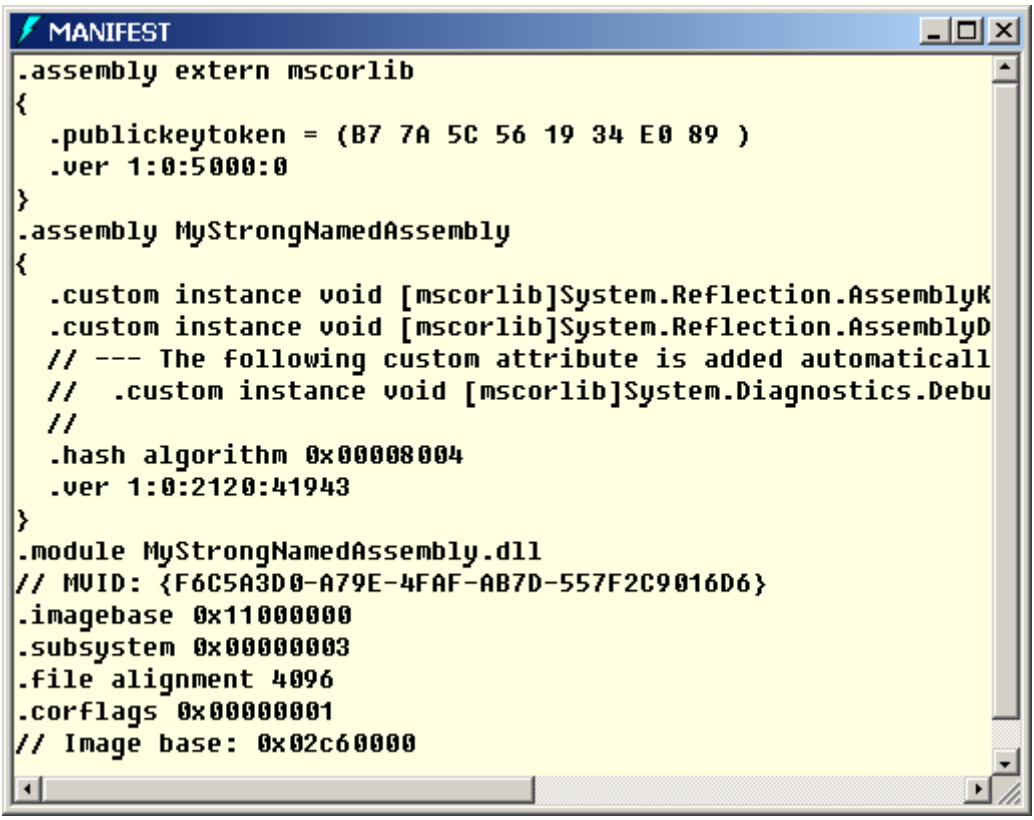
Първо ще създадем двойка публичен / частен ключ (public/private key pair). За целта използваме Strong Name инструмента (**sn.exe**), който е част от .NET Framework SDK:

```
sn -k keypair.snk
```

Тази команда генерира по случаен начин двойка публичен/частен ключ и ги записва във файла **keypair.snk**.

След като вече имаме двойката ключове, можем да пристъпим към създаване на силно именувано асембли:

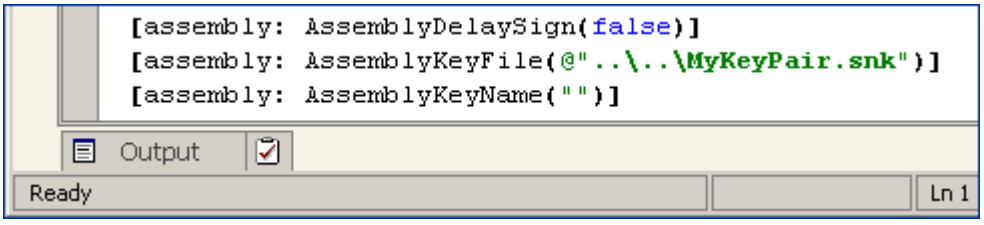
1. Стартираме Visual Studio .NET 2003.
2. Създаваме нов проект от тип Class Library: File -> New -> Project -> Visual C# Projects -> Class Library. Задаваме име на проекта **MyStrongNamedAssmbly** и потвърждаваме с бутона **[OK]**.
3. Компилираме проекта. Създадохме обикновено асембли (то се намира в поддиректорията **bin\Debug** на проекта, във файла **MyStrongNamedAssembly.dll**). Неговият манифест изглежда по следния начин:



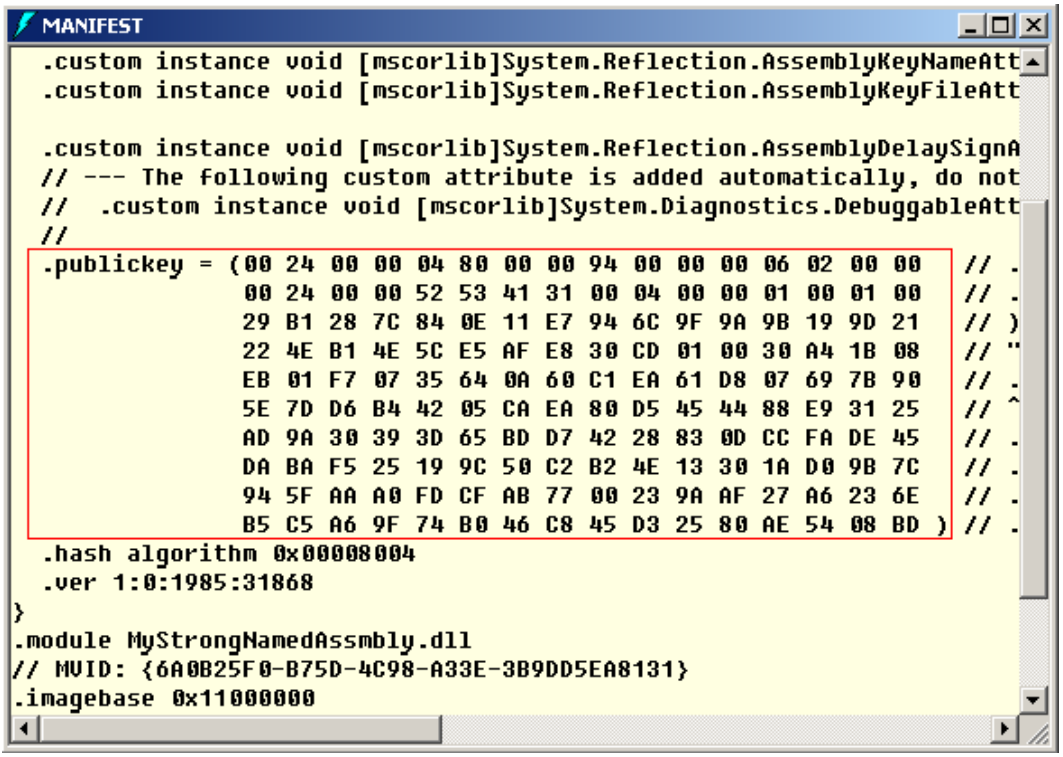
4. Копираме създадения по-горе файл **keypair.snk** в папката, която съдържа VS.NET проекта (**MyStrongNamedAssembly.csproj**).

5. От Solution Explorer отваряме файла **AssemblyInfo.cs** и намираме реда **[assembly: AssemblyKeyFile("")]**. Това е атрибутът, който ще използваме, за да посочим двойката публичен/частен ключ. Конструкторът на **AssemblyKeyFileAttribute** приема като параметър име на файл, съдържащ двойката ключове. Може да бъде твърд път (**c:\keyfiles\keypair.snk**) или относителен път спрямо **obj\Debug** поддиректорията на проекта. В нашия случай можем да зададем относителен път: **..\..\keypair.snk**.

6. Променяме атрибута **AssemblyKeyFileAttribute** на **[assembly: AssemblyKeyFile(@"..\..\keypair.snk")]**:



7. Компилираме проекта отново. Сега вече създадохме силно именувано асембли (то се намира отново в поддиректорията **bin\Debug** на проекта, във файла **MyStrongNamedAssembly.dll**). Неговият манифест се различава от предходния по **.publickey** частта:



Конфигурационни файлове в .NET Framework

Конфигурационните файлове в .NET Framework са текстови файлове в XML формат и служат за задаване на различни настройки на .NET приложенията. Съществуват няколко вида конфигурационни файлове:

1. Конфигурационен файл за настройките на машината – **Machine.config** – този файл се намира в **%runtime install path%\Config** (например **C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\Config**) и съдържа настройки оказващи влияние върху CLR за локалния компютър.

2. Конфигурационни файлове на приложенията – съдържат настройки, специфични за дадени приложения. Те са два вида: за уеб-базирани приложения се казват винаги **Web.config** и се намират в коренната директория на уеб приложението или уеб услугата в Internet Information Server и за Windows-базирани приложения – образуват се от името на приложението с **.config** разширение (например: ако имаме приложение **MyLibrary.dll**, конфигурационният му файл ще се казва **MyLibrary.dll.config**).

3. Publisher Policy File – указват на всички приложения да използват по-нова версия на външно асембли от тази, спрямо която са били компилирани (version redirect). По-нататък ще разгледаме как се използват (вж. Създаване на Publisher Policy File).
4. Конфигурационни файлове за сигурността (security policy) – съдържат описание на правата за изпълнение на инсталираните асемблита. В .NET Framework съществуват няколко нива на сигурност:
   - ниво организация
   - ниво машина
   - ниво потребител

Следващата таблица показва тяхното местоположение в зависимост от операционната система:

Enterprise security policy configuration file	
Windows 2000, XP, 2003	`%runtime install path%\Config\ Enterprisesec.config`
Windows NT	`%runtime install path%\Config\ Enterprisesec.config`
Windows 98 and Windows Millennium Edition (Windows Me)	`%runtime install path%\Config\ Enterprisesec.config`

Machine security policy configuration file	
Windows 2000, XP, 2003	`%runtime install path%\Config\ Security.config`
Windows NT	`%runtime install path%\Config\ Security.config`
Windows 98 and Windows Me	`%runtime install path%\Config\ Security.config`
User security policy configuration file	
Windows 2000, XP, 2003	`%USERPROFILE%\Application Data\ Microsoft\CLR security config\ vxx.xx\Security.config`
Windows NT	`%USERPROFILE%\Application Data\ Microsoft\CLR security config\ vxx.xx\Security.config`
Windows 98 and Windows Me	`%WINDIR%\username\CLR security config\vxx.xx\Security.config`

Всички тези конфигурационни файлове са важни за разпространението на .NET приложенията, на което ще се спрем след малко.

Как CLR намира асемблитата?

Важно е за разработването и разпространението на .NET приложения да се познава как CLR търси асемблитата, които дадено приложение изисква, за да се изпълни. По подразбиране CLR се опитва да намери асемблитата със същата версия, с която приложението е било компилирано. Когато .NET приложение изиска външно асембли, се изпълняват следните стъпки:

1. Определя се вярната версия на нужното асембли – чрез проверяване на конфигурационните файлове (за настройките на машината, на приложението и publisher policy file).
2. Проверява се дали приложението е използвало асембли със същото име. В такъв случай се зарежда последното използвано асембли.
3. Проверява се Global Assembly Cache. Ако асембли със същото име се намира там, се използва то.
4. Изпълнява се търсене на асембли (assembly probing) чрез следните стъпки:
   - Ако конфигурационните файлове не променят версията на изискваното асембли, тогава CLR се опитва да налучка местоположението му като се базира на неговото име.
   - Ако е намерен **<codebase>** елемент в конфигурационните файлове се търси само в пътя, посочен там. Ако асемблито не е намерено, се регистрира грешка и се прекратява търсенето.
   - Търси се в поддиректориите, посочени в **<probing>** секцията на конфигурационния файл на приложението. Ако не е намерено асемблито, се прави заявка към Windows Installer да инсталира изискваното асембли. Тази възможност на Windows Installer се нарича инсталиране при заявка (install-on-demand).

За асемблита, които не са силно именувани, CLR не проверява GAC за тяхното наличие и не проверява версията.

Пример 1: Търсене на асембли (probing)

За да поясним описания процес, ще дадем един пример. Нека имаме Windows-базирано приложение **BaseDir\MyApp.exe**, което използва ресурси от асембли **MyLibrary**, което не е силно именувано. Конфигурационният файл **MyApp.exe.config** съдържа:

MyApp.exe.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <runtime>
```

```
      <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
         <probing privatePath="bin;bin2\subdir"/>
      </assemblyBinding>
   </runtime>
</configuration>
```

При стартиране на **MyApp.exe** асемблито **MyLibrary** се търси в следните директории:

```
BaseDir\MyLibrary.dll
BaseDir\MyLibrary\MyLibrary.dll
BaseDir\bin\MyLibrary.dll
BaseDir\bin\MyLibrary\MyLibrary.dll
BaseDir\bin2\subdir\MyLibrary.dll
BaseDir\bin2\subdir\MyLibrary\MyLibrary.dll
(после същите файлове, но с разширение .exe)
```

С помощта на инструмента Assembly Binding Log Viewer (**Fuslogvw.exe**), който е част от .NET Framework SDK, може да се разгледа детайлно в кои директории и в какъв ред CLR търси асемблитата.

Пример 2: Търсене на асембли с тага <codebase>

Нека разширим малко предходния пример. Добавяме в конфигурационния файл **MyApp.exe.config** таг **<codebase>** в частта между таговете **<dependentAssembly>** и **<dependentAssembly>**.

MyApp.exe.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <runtime>
       <assemblyBinding xmlns="urn:schemas-microsoft-
com:asm.v1">
       <dependentAssembly>
          <assemblyIdentity name="MyLibrary"/>
          <codeBase version="2.0.0.0"
            href="CodeBase\MyLibrary.dll" />
       </dependentAssembly>
       <probing privatePath="bin;bin2\subdir;Lib1"/>
      </assemblyBinding>
   </runtime>
</configuration>
```

При стартиране на **MyApp.exe** асемблито **MyLibrary** се вече търси в само в посочената директория:

```
CodeBase\MyLibrary.dll
```

Забележка: CLR дори не опитва да намери асембли **MyLibrary.exe**.

При посочен таг **<codebase>** CLR търси асембли само с посочено име и само в посочената директория. Ако не бъде намерено асемблито, търсенето се прекратява и се съобщава за грешка.

Създаване на Publisher Policy File

Файловете с политика на издателя (publisher policy file) са специален вид конфигурационни файлове, които се компилират и инсталират в Global Assembly Cache и указват на всички приложения да използват по-нова версия на външно асембли от тази, спрямо която са били компилирани (version redirect).

Използването на такива файлове можем да демонстрираме чрез няколко стъпки:

1. Създаваме файл с име **pubPolicy.config** (с Notepad или Visual Studio .NET – няма предварително зададени шаблони с Visual Studio .NET). Целта на този файл е да укаже на CLR при извикване на асемблито с посочения манифест (име - **myRedirectedAssembly**, публичен ключ - **32ab4ba45e0a69a1**, версия - **1.0.0.0**) да се зареди асембли със същото име и публичен ключ, но версия **2.0.0.0**. Понеже този файл указва политика на зареждане на асемблита той се нарича publisher policy file. Като съдържание въвеждаме следното:

pubPolicy.config

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="myRedirectedAssembly"
          publicKeyToken="32ab4ba45e0a69a1" />
        <!-- Redirecting to version 2.0.0.0 -->
        <bindingRedirect oldVersion="1.0.0.0"
          newVersion="2.0.0.0"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

2. Компилираме на publisher policy file до publisher policy assembly. Ще използваме създадения файл, за да създадем асембли и да го именуваме силно (ще използваме същата двойка публичен / частен ключ, която създадохме в точката Създаване на силно именувано асембли. Това става с помощта на инструмента Assembly Linker (**al.exe**):

```
al /link:pubPolicy.config
/out:policy.1.0.myRedirectedAssembly.dll /keyfile:keypair.snk
```

Изходът от тази команда е асембли, записано във файл с име **policy.1.0.myRedirectedAssembly.dll**.

3. Добавяне в GAC. Необходимо е така създаденото Publisher Policy Assembly да бъде добавено в Global Assembly Cache. Това става със следната команда:

```
gacutil /i policy.1.0.myRedirectedAssembly.dll
```

(За повече информация вж. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconcreatingpublisherpolicyfile.asp).

Global Assembly Cache

Всяка машина (персонален компютър, мобилно устройство и др.), на която е инсталиран .NET Framework, има област наречена Global Assembly Cache (GAC). GAC е специално проектиран да съдържа асемблита, които могат да се ползват от няколко различни приложения. Целта е общите компоненти от различни софтуерни продукти да не се дублират, а да се публикуват като общи и да са достъпни за всички приложения. Пример за такъв общ компонент може да бъде библиотека за графичен потребителски интерфейс, която доставя съвкупност от графични контроли. Такива контроли могат да се използват в много приложения и не е необходимо всяко от тях да инсталира библиотеката поотделно.

За да се изяснят по-добре целта и предимствата на GAC, ще опишем първо две други понятия - DLL адът (DLL Hell) и side-by-side execution.

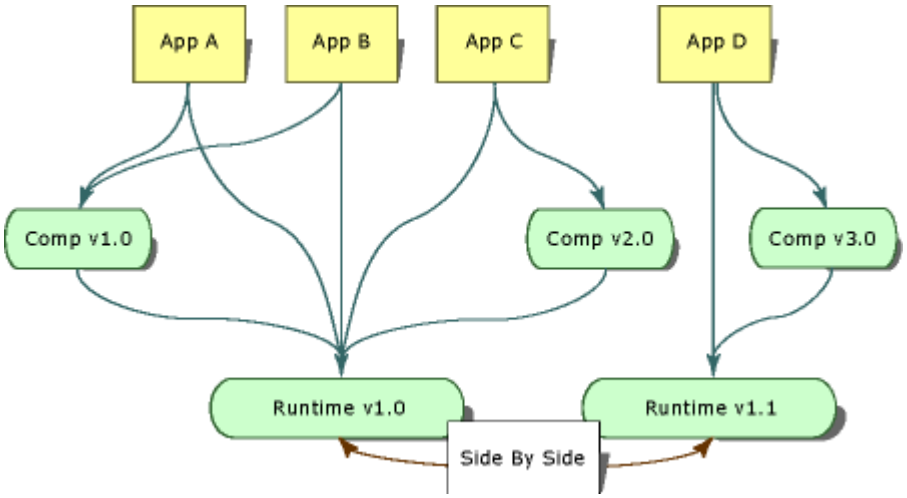
DLL адът (DLL Hell)

DLL ад се нарича ситуацията, в която инсталацията на дадено приложение заменя съществуващ файл (**.DLL**, **.OCX** или **.VBX**) с по-стара версия или нова версия, която не е съвместима с другите версии. В такъв случай приложения, използващи ресурси от въпросния файл, спират да работят. Последното инсталирано приложение работи, но всички останали, които използват ресурси от въпросния файл, не работят коректно. Потребителят, който не е длъжен да познава тази ситуация, хвърля вината върху производителя на (вече) неработещия софтуер.

Side-by-side execution

Side-by-side execution се нарича възможността едновременно да съществуват и да се използват приложения, компоненти и дори CLR с различни версии на един компютър. Функцията на GAC е да намери и зареди асемблито със посочената версия, дори и ако тя не е последна.

Следващата графика показва компютър, на който има инсталирани версии 1.0 и 1.1 на .NET Framework, четири приложения (A-D), които използват, различни версии на компонента `Comp`. Версии 1.0 и 2.0 на `Comp` използват .NET Framework 1.0, докато версия 3.0 използва .NET Framework 1.1.



Предимства и недостатъци на GAC

Инсталирането на асемблита в GAC има следните предимства:

- Единствено място за инсталиране на асемблита, които се използват от множество приложения. Много по-лесно е за инсталиране на нова версия на даден компонент в GAC и създаване на publisher policy file за пренасочване всички приложения да използват новата версия, отколкото да се търсят всички копия в частните папки на различните приложения.
- По-добро бързодействие при цифрово подписаните асемблита – това е така, защото проверката на цифровия подпис се прави веднъж – при инсталиране в GAC, докато при частните асемблита се прави на всяко зареждане.
- По-добро бързодействие и спестяване на ресурси при зареждане на няколко копия от едно и също асембли – в такъв случай CLR просто пренасочва заявките към вече зареденото асембли, вместо да заделя памет и да зарежда асембли в паметта отново.
- Решава проблема с DLL ада и позволява side-by-side execution.

Използването на GAC има и своите недостатъци:

- Асемблитата, които се инсталират в GAC трябва задължително да са силно именувани.
- GAC се намира в `%systemroot%\assembly` и като поддиректория на Windows се изискват администраторски права, за да се променя съдържанието му.

- Директно копиране в GAC не е допустимо – асемблитата трябва да се инсталират посредством някой от следните начини:
  - Чрез Windows Installer 2.0.
  - Чрез инструмента **gacutil.exe** от .NET Framework SDK.
  - Чрез Windows Shell разширението за Windows Explorer, съдържащо се в **SHFusion.dll**.
  - Чрез административната конзола на .NET Framework – **mscorcfg.msc**.
  - Чрез API функциите на класа **Microsoft.CLRAdmin.Fusion** от .NET Framework библиотеката **mscorcfg**.

Работа с GAC – пример

В следващия пример ще използваме силно именуваното асембли **MyStrongNamedAssembly.dll** от частта "Създаване на силно именувано асембли", за да илюстрираме работата с GAC:

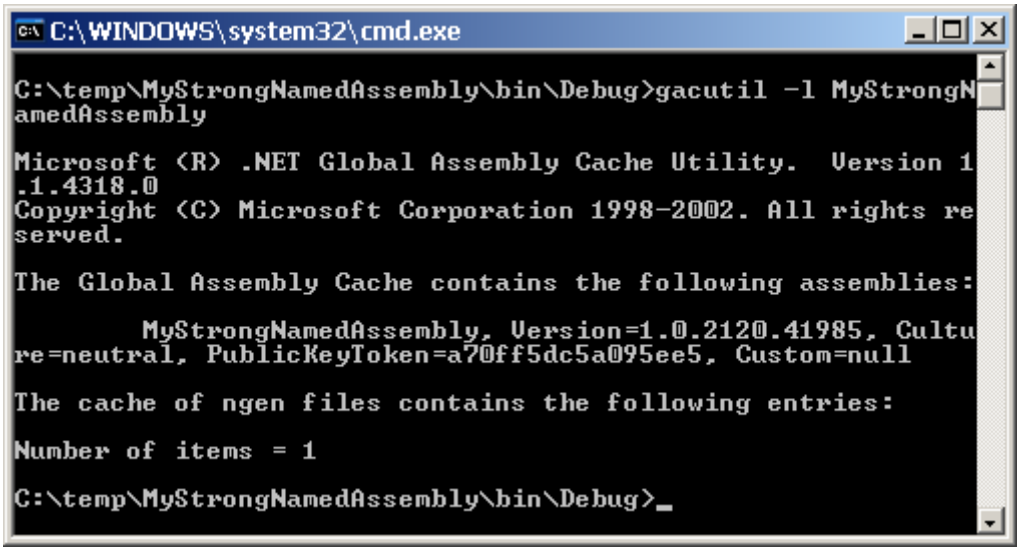
1. Добавяме асемблито в GAC със следната команда:

```
gacutil /i MyStrongNamedAssembly
```

2. За да видим съдържанието на GAC използваме командата:

```
gacutil /l MyStrongNamedAssembly
```

Резултатът от тази команда трябва да е подобен на следния:



3. За дерегистрация (изтриване) на асембли от GAC използваме следната команда:

```
gacutil /u MyStrongNamedAssembly
```

Разпространение и инсталиране на програмни пакети

След като .NET приложението е преминало успешно през фазите на разработване и тестване, трябва да се създаде инсталационен пакет (Windows Installer Package, MSI package), който ще се използва от потребителите на приложението. Много често този етап се подценява от софтуерните разработчици, но той е изключително важен от гледна точка на клиента. Инсталационният пакет е първият досег на клиента със софтуерния продукт и затова той създава първото впечатление. Към инсталационния пакет има много изисквания но най-важните от тях са следните:

- да поставя на клиентската машина всичко необходимо за нормалната употреба на закупения софтуер, без да се пречи на изпълнението на вече инсталирания софтуер;
- да бъде лесен за използване;
- да предоставя възможност за деинсталиране като възстановява машината до предишното й състояние.

Докато третото изискване е сравнително лесно за постигане със съществуващите технологии и второто зависи от сложността на решението и аудиторията, към която е насочено приложението, то за първото е необходима предварителна подготовка. Това се обуславя от факта, че различните типове приложения се инсталират по различен начин.

В зависимост от типа на приложението инсталационният пакет включва различни комбинации от съставните части на приложението: файлове и папки, асемблита, инсталационни компоненти, COM обекти, сървърни компоненти, бази данни, настройки на Internet Information Server и т. н.

Файлове и папки

Съществуват много различни видове файлове, които могат да бъдат инсталирани заедно с приложението. Типовете файлове зависят от големината, целта, вида и сложността на приложението. Следващата таблица показва най-често използваните типове файлове, които се разпространяват с .NET базираните приложения:

Тип на файла	Windows приложение/услуга	Уеб приложение/услуга
изпълними файлове (`.exe`)	**X**	**X**
динамични библиотеки (`.dll`)	**X**	**X**
конфигурационни файлове (`.config`)	**X**	**X**

бази от данни	**X**	**X**
уеб страници (`.htm`**,** `.html` **и др.)**		**X**
уеб форми (`.aspx`**,** `.ascx` **и др.)**		**X**
файлове за уеб услуги (`.asmx`**,** `.disco`**, ...)**		**X**
XML, XSD файлове	**X**	**X**
други необходими файлове	**X**	**X**

Конфигурационни файлове

Както вече описахме, конфигурационните файлове съдържат определени настройки, влияещи върху изпълнението на програмата. Най-големият проблем при разпространение на конфигурационните файлове е тяхното управление при инсталиране в различни обкръжаващи среди.

Например, ако в даден проект се разработват Windows приложение и уеб услуга, то в конфигурационния файл на Windows приложението ще се съхранява URL до тази уеб услуга. URL адресът обаче ще е различен при различните етапи на разработване, тестване, интегриране на системата и при реално функциониращa среда. Необходимостта от различни конфигурационни файлове се отежнява и от факта, че за отдалеченото извикване (.NET Remoting) са необходими допълнителни настройки в конфигурационните файлове, които коренно се различават в различните среди и трябва да се тестват цялостно при използване на .NET Remoting технологията.

Използване на конфигурационни файлове

За да се управляват по-лесно различите конфигурации, може да се използва следното решение:

1. Добавяме конфигурационен файл, необходим за средата за разработка към проекта във Visual Studio .NET (**`App.config`** или **`Web.config`** в зависимост от типа на проекта). Visual Studio .NET автоматично добавя **`Web.config`** при създаване на ново ASP.NET приложение.
2. Създаваме отделен конфигурационен файл за всяка среда, където настройките се различават. Именуваме ги така, че от името им да става ясно за каква среда са необходими (напр. **`Test.Web.config`**, **`Production.Web.config`**) и ги добавяме към проекта.
3. Включваме само необходимия конфигурационен файл в зависимост от средата, в която ще се инсталира, и го преименуваме на изискваното име като част от инсталационния процес. Изискваните имена са **`<AppName>.exe.config`** за Windows приложения или **`Web.config`** за уеб приложения.

Процесът на инсталиране на конфигурационен файл в тестова среда е подобен на този за реално функциониращa среда. Ако, обаче, се инсталират различни конфигурационни файлове, тогава файлът за реално функциониращата среда няма да премине необходими тестове (ще остане нетестван).

Един от начините за решаване на този проблем е да не се създават различни файлове за тестовата и реалната среда. Ако са необходими допълнителни настройки в конфигурационния файл за тестване, тогава тестващият екип може да ги променя преди да започне тестовете.

Файлове за уеб услуги

Много от проблемите, засягащи инсталирането на уеб приложенията се отнасят и за уеб услугите – настройки на IIS, инсталиране на HTTP модули и обработчици (HTTP handlers).

Малка разлика между уеб приложенията и уеб услугите представлява видът на файловете. За уеб приложенията се инсталират предимно **`.aspx`** и **`.ascx`** файлове, докато уеб услугите изискват **`.asmx`** и **`.disco`** файлове.

XSD файлове

XSD файловете са документи, които служат за дефиниране и проверка на съдържанието и структурата на XML документи. Инсталиране на XSD файлове е необходимо само ако приложението има нужда от тях по време на изпълнението си. Например, ако метод разработен в уеб услуга връща като резултат силно типизиран **`DataSet`**, тогава XSD файлът трябва да се инсталира (заедно с ASMX и DISCO файловете), тъй като уеб услугата ще съдържа референция към него. Когато разработчиците разглеждат DISCO файла на тази услуга от Visual Studio .NET те ще могат да разгледат и схемата чрез връзката **View Schema**.

J# инсталационен пакет (J# Redistributable Package)

Инсталация на Visual J# .NET Redistributable Package (**`vjredist.exe`**) е необходима, ако приложението използва компоненти или библиотеки написани на Visual J#, понеже J# не е част от .NET Framework, а е допълнително разширение.

Асемблита

Инсталирането на многомодулни асемблита изисква наличието на всички модули. В противен случай използването на асембли с липсващ модул ще предизвика грешка.

Силно именувани асемблита

Когато .NET приложение създава референция към силно именувано асембли, CLR изисква да бъде включена неговата версия. При стартиране на приложението CLR ще търси асембли с посочената версия. Това

означава, че ако асембли с по-нова версия бъде копирано върху файла със старата версия, приложението ще спре да работи.

Едно от възможните решения е инсталиране на силно именуваното асембли в GAC. По-късно, при разработване на асембли с по-висока версия се разработва и Publisher Policy File, който описахме в точката за Конфигурационни файлове. Новото асембли и Publisher Policy File също се инсталират в GAC. По този начин всички приложения започват да използват новата версия на асемблито.

Частни асемблита

Ако дадено асембли е проектирано да се използва само от едно приложение, то трябва да се инсталира като частно асембли. Частните асемблита обикновено се инсталират в папката на приложението. Други подходящи местоположения на частните асемблита са:

- Директорията, съдържаща изпълнимия файл или нейна поддиректория.
- **`Bin`** папката, намиращата се във виртуалната директория на уеб приложението или уеб услугата в IIS.
- Местоположение описано с **`<codebase>`** елемент в конфигурационния файл на приложението.

В повечето случаи частните асемблита следва да се инсталират в директорията, съдържаща изпълнимия файл, понеже това е мястото откъдето CLR започва търсенето.

Частните асемблита се използват само от приложението, с което са инсталирани. По този начин приложението е изолирано от останалия софтуер на машината. Това е един от начините да се избегнат проблемите с различните версии – т. нар. "DLL ад".

Споделени асемблита

В някои случаи едно и също асембли може да се използва от няколко приложения. Тогава съществуват следните възможности за инсталация:

- Инсталиране на асемблито с всяко приложение като частно асембли – по този начин всяко копие е абсолютно независимо от останалите. Инсталирането на множество копия поражда определени проблеми, които следва да се вземат под внимание: възможно е да съществуват множество версии на асемблито, които дори използват различни версии на .NET Framework; при наличието на определени проблеми с асемблито, които са отстранени в следващи версии се създават трудности със замяната на всички копия на машината.
- Инсталиране на асемблито в една обща директория – за всяко от използващите го приложения се посочва тази директория с **`<codebase>`** елемента в конфигурационния файл. Ако се използва една директория за всички споделени асемблита на дадена компа-

ния, се загубва възможността за side-by-side execution, но се решават проблемите с версионизиране на асемблитата (за да е възможен ъпгрейд на асемблита с просто копиране те не трябва да бъдат силно именувани). Трябва да се планира внимателно избор на подобна стратегия, защото, ако на следващ етап възникне изискване за използване на side-by-side execution, стратегията трябва да се смени, което ще доведе до трудности.

- Инсталиране в GAC – препоръчително е при инсталиране на асембли в GAC да се изпълнява броене на референциите, за да се избегне премахването на асембли от GAC, докато все още съществува приложение на машината, което го използва. Съществуват два начина да се постигне това: чрез Windows Installer или чрез **`gacutil.exe`** (използват се параметри **`/ir`** за инсталиране и **`/ur`** за деинсталиране чрез броене на референциите).

Потребителски контроли тип Windows Forms (Windows Forms User Controls)

Потребителските контроли тип Windows Forms са асемблита, които могат да се извикват от уеб страници като се свалят на клиентската машина и се стартират локално на нея. По подразбиране такива контроли работят с намалени права, за да не застрашават сигурността на потребителя.

Инсталирането на този тип контроли поставя известни предизвикателства, понеже те се изпълняват в контекста на правата, дадени им от уеб браузъра и от политиките за сигурност на .NET Framework.

От гледна точка на сигурността съществуват два вида потребителски контроли тип Windows Forms – такива, които се стартират с права по-подразбиране, и такива, които изискват по-високи права.

Контролите, които изискват по-високи права за изпълнение следва да бъдат силно именувани (strong-named) или цифрово подписани (digitally signed). Това позволява на администраторите да разрешат по-високи права на асемблита от дадена компания, без да компрометират сигурността.

Изискването за промяна настройките на сигурността прави контроли от този тип да бъдат по-лесни за използване в корпоративна Интранет среда, където администраторите на домейни централизирано могат да променят тези настройки. За да бъдат контролите използваеми през Интернет, потребителите трябва да следват определени инструкции, за да променят настройките на сигурността.

Добра практика е създаване на инсталационен пакет с Windows Installer, който прави необходимите промени в сигурността. Това спестява на потребителите следване на сложни инструкции, които са непонятни за тях. (Въпреки това промените, които се правят от инсталационния пакет би следвало да бъдат добре

	обяснени.)

Инсталационни компоненти

Разпределените .NET приложения се състоят не само от традиционни програмни файлове и асемблита, но и от допълнителни ресурси като опашки от съобщения (message queues), дневник на събитията (event logs), индикатори за производителността (performance counters) и бази данни (databases). Тези ресурси са необходими за нормалното функциониране на приложенията и като такива следва да бъдат създадени по време на инсталационния процес.

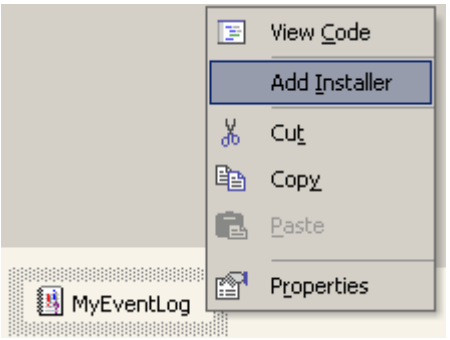
.NET Framework предоставя компоненти, които създават и конфигурират ресурсите по време на инсталация и ги изтриват по време на деинсталация. Тези компоненти се интегрират с инсталационните инструменти (**`InstallUtil.exe`**) на Windows Installer и Windows Installer Service. Съществуват два типа инсталационни компоненти: предварително създадени инсталационни компоненти и инсталационни класове (Installer Classes).

Предварително създадени инсталационни компоненти

Майкрософт предоставят пет предварително дефинирани инсталационни компоненти, които могат да бъдат използвани при създаване на инсталации за приложения:

- **`EventLogInstaller`** – използват се при създаване и настройване на дневник на събитията (event logs).
- **`MessageQueueInstaller`** – използват се при създаване и настройване на опашки от съобщения (message queues).
- **`PerformanceCounterInstaller`** – използват се при създаване и настройване на индикаторите за производителността.
- **`ServiceInstaller`** и **`ServiceProccessInstaller`** – използват се при създаване и настройване на Windows услуги.

Всеки един от тези класове се използва, за да се инсталират необходимите ресурси по време на инсталационния процес. След добавяне на определен компонент в дизайн изгледа на Visual Studio .NET има възможност за генериране на съответния инсталационен клас чрез контекстното меню на компонента, както е показано на картинката:



При добавяне на инсталационния компонент към проекта се създава клас, наречен **`ProjectInstaller`**, който съдържа инсталационните класове на всички компоненти използвани.

По време на инсталационния процес се задейства **`ProjectInstaller`**, който изпълнява инсталация за всеки от съдържащите компоненти.

Инсталационни класове (Installer Classes)

В някои случаи се налага да се инсталират ресурси, за които няма предварително създадени инсталационни компоненти. Може да се наложи да се изпълнят някои специфични действия по време на инсталационния процес. Добър пример за това е създаване на база данни, попълване на някои таблици и прекомпилиране на дадено асембли до машиннозависим (native) код (чрез инструмента **`NGen.exe`**) след успешно приключване на инсталацията. За тази цел могат да се използват класовете от пространството **`System.Configuration.Install`** и най-вече класа **`Installer`**, който се наследява, за да се имплементира потребителски инсталатор за даден компонент.

COM базирани обекти

Възможно е в бъдеще компонентите, изградени с помощта на .NET Framework да заменят изцяло компонентите базирани на Component Object Model (COM, COM+, DCOM – за повече информация виж http://www.microsoft.com/com). Докато настъпи този момент, обаче, ще трябва да имаме предвид тяхното съществуване и да им отделим подходящото внимание.

За да бъдат използваеми всички COM базирани компоненти, трябва да бъдат регистрирани на компютъра и да са инсталирани в COM+ каталога.

Главното предизвикателство в разпространението на COM базирани компоненти е в това да бъдем сигурни, че те могат да комуникират с асемблитата на приложението, както и асемблитата да могат да бъдат извиквани през COM.

При инсталиране на COM базиран компонент трябва да бъде извършена регистрация в COM+ каталога (чрез **`regsvr32.exe`** или Windows Installer) и .NET приложението, което го използва трябва да има достъп до Interop

асемблито, съдържащо дефиниции на типовете в COM обекта (генерира се чрез **tlbimp.exe**).

При инсталиране на асемблита, които ще се извикват през COM трябва да се извърши регистрация чрез .NET Framework SDK инструмента Assembly Registration Tool (**regasm.exe**).

При инсталирането на асемблита, които ще се извикват през COM имайте предвид, че ако асемблито няма да се инсталира в GAC, трябва да се използва параметъра **/codebase** при регистрация с **Regasm.exe** (или съответното настройване при Windows Installer, както ще видим в примера по-долу). В противен случай COM няма да може да открие местоположението на асемблито.

Сървърни компоненти (Serviced Components)

Сървърните компоненти, създадени с .NET Framework, разчитат на COM за предоставянето на компонентни услуги като управление на транзакциите (transaction management), пулинг на обекти (object pooling) и като цяло предоставяне на услуги, които се обуславят от COM технологията.

Сървърните компоненти се стартират като COM базирано приложение и затова те трябва да регистрират в COM каталога. Това налага следните инсталационни и конфигурационни изисквания за компонентните услуги:

- Асемблитата трябва да са силно именувани.
- Асемблитата трябва да са регистрирани в регистрите на Windows (Windows registry).
- Библиотеката с типовете (type library) трябва да е регистрирана при клиента.

Динамична регистрация на сървърни компоненти

Сървърните компоненти често се регистрират динамично първия път, когато приложение се опита да ги използва. Тогава CLR регистрира асемблито, библиотеката с типовете и конфигурира COM+ каталога. Регистрация се прави само веднъж за дадена версия на асембли. Това е най-лесният начин за регистриране на сървърни компоненти, но за да е успешен процесът, който ги стартира, трябва да има административни права върху машината. Допълнително ограничение е фактът, че динамична регистрация е възможна само ако асемблито се извиква от управляван код. Ако извикването се прави от неуправляван код, динамичната регистрация не е възможна.

В много случаи процесът, който извиква асемблито, няма изискваните права, за да извърши динамична регистрация. Пример за това са компоненти, които се извикват от уеб приложение. Такива компоненти не могат да бъдат регистрирани, защото ASP.NET процесът няма административни

права (освен, ако не е настроен да се изпълнява в контекста на потребителя **SYSTEM**, което не е добра идея и компрометира сигурността). В такъв случай регистрацията ще се провали ще се предизвика грешка "отказан достъп" (access denied). Затова трябва да направим необходимата регистрация по време на инсталационния процес.

Регистрация на сървърни компоненти при инсталация

Понеже сървърните компоненти използват предимствата на COM+ услугите, те имат същите изисквания относно инсталацията:

- Комуникацията със сървърните компоненти се осъществява през DCOM по подразбиране. Поради тази причина Interop асемблитата трябва да се регистрират на клиентската машина.
- В допълнение към COM настройките, които се правят декларативно при разработката на асемблито с помощта на атрибути, трябва да осигурят, че конфигурационните настройки (като присъединяване на потребители към потребителски групи, настройване на подходящи права за изпълнение, задаване в кой контекст на сигурност (security context) ще се изпълнява процесът) са подходящо създадени и компонентът е готов за използване. Това може да бъде постигнато чрез добавяне на скрипт към допълнителните действия (custom actions) на Windows Installer, който да регистрира компонента.

Възможно е настройването на COM приложението да използва SOAP вместо DCOM. Това заобикаля изискването да се инсталират Interop асемблита на клиентската машина. Този метод не позволява да се използва създадената от DCOM транзакция клиент-сървър и трябва да се добави допълнителен код, който да управлява транзакциите.

Настройки на Internet Information Server (IIS)

ASP.NET приложенията изискват наличието на Internet Information Server (IIS), за изпълнението си. В някои случаи е необходимо да се правят промени по настройките на IIS, за да се изпълни приложението. При инсталиране на ASP.NET приложения на Windows Server 2003, те се изпълняват от IIS 6.0. При по-стари версии на Windows, се използва IIS 5.0 или 5.1.

IIS 6.0 позволява два режима на изолация на приложенията:

- изолация на работния процес (по подразбиране)
- IIS 5.0 режим на изолация

IIS 6.0 и изолация на работния процес

Когато ASP.NET приложенията работят в режим на изолация на работния процес, те работят в процеса **W3wp.exe**. В този случай моделът на процесите, който е вграден в ASP.NET е изключен и се използва архитектурата

на изолация на работния процес от IIS 6.0. С този режим на изолация и с помощта на Application pools може да се изолира всичко (от дадено уеб приложение до множество сайтове) в собствен процес на WWW услугата (WWW Publishing Service), като по този начин дадено проблемно приложение не може да повлияе негативно на останалите.

Процесите, изпълняващи отделните приложения, са напълно разделени от основната WWW услуга – **Inetinfo.exe**. ISAPI приложенията (Internet Server Applications) също работят напълно отделно от WWW услугата, което предпазва всички сайтове на сървъра от грешки в изпълнението им. Ако възникне грешка в ISAPI приложение, само процесът, който съдържа проблемното ISAPI, е засегнат.

Работният процес може да бъде конфигуриран да използва определен процесор, което позволява по-голям контрол върху разпределението на системните ресурси. Като допълнение уеб приложенията се стартират в контекста на **Network Service** потребителя, който има по-малки привилегии за достъп от **LocalSystem**. Това води до повишаване на сигурността.

IIS 5.0 и режим на изолация

Ако се използва изолационният модел на IIS 5.0, ASP.NET се изпълнява в собствения модел на процесите (**Aspnet_wp.exe**) и използва собствените конфигурационни настройки. Зададеният изолационен модел е на ниво сървър и важи за всички приложения, използващи IIS.

IIS 5.0 режим на изолация трябва да се използва от приложения, които влизат в конфликт с режима на изолация на работния процес. Следните характеристики създават посочения конфликт:

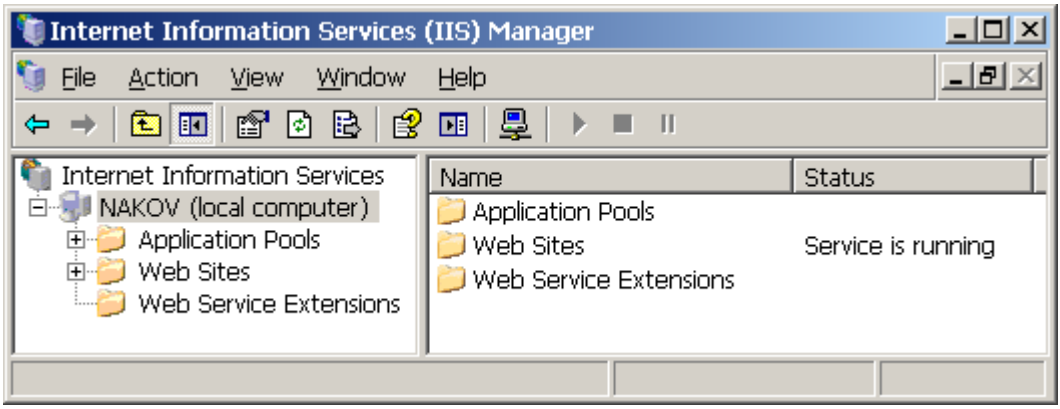
- Зависимост от **Inetinfo.exe**. Ако приложението изисква да се стартира в контекста на процеса **Inetinfo.exe**, тогава трябва се използва IIS 5.0 режим на изолация.
- Изисква се използването на Read Raw Data Filters. Тяхното използване изисква IIS 5.0 режим на изолация.
- Изисква **Dllhost.exe**. Приложенията, които трябва да се изпълняват в обкръжението на **Dllhost.exe** могат да бъдат изпълнени само в IIS 5.0 режим на изолация.

Ако е необходимо изпълнението на приложение, което не изпълнява изискванията за изпълнение в режим на изолация на работния процес, трябва да се премине към IIS 5.0 режим на изолация. В такъв случай не могат да се използват предимствата на изолацията на работния процес от IIS 6.0.

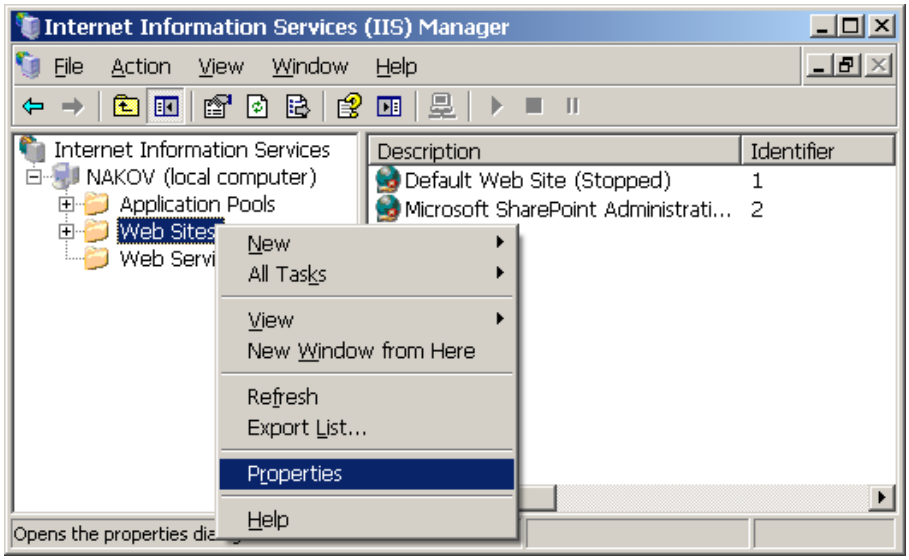
Конфигуриране на изолационния режим

Изолационният режим може да се конфигурира от административната конзола на IIS по следния начин (примерът е с Windows 2003 Server):

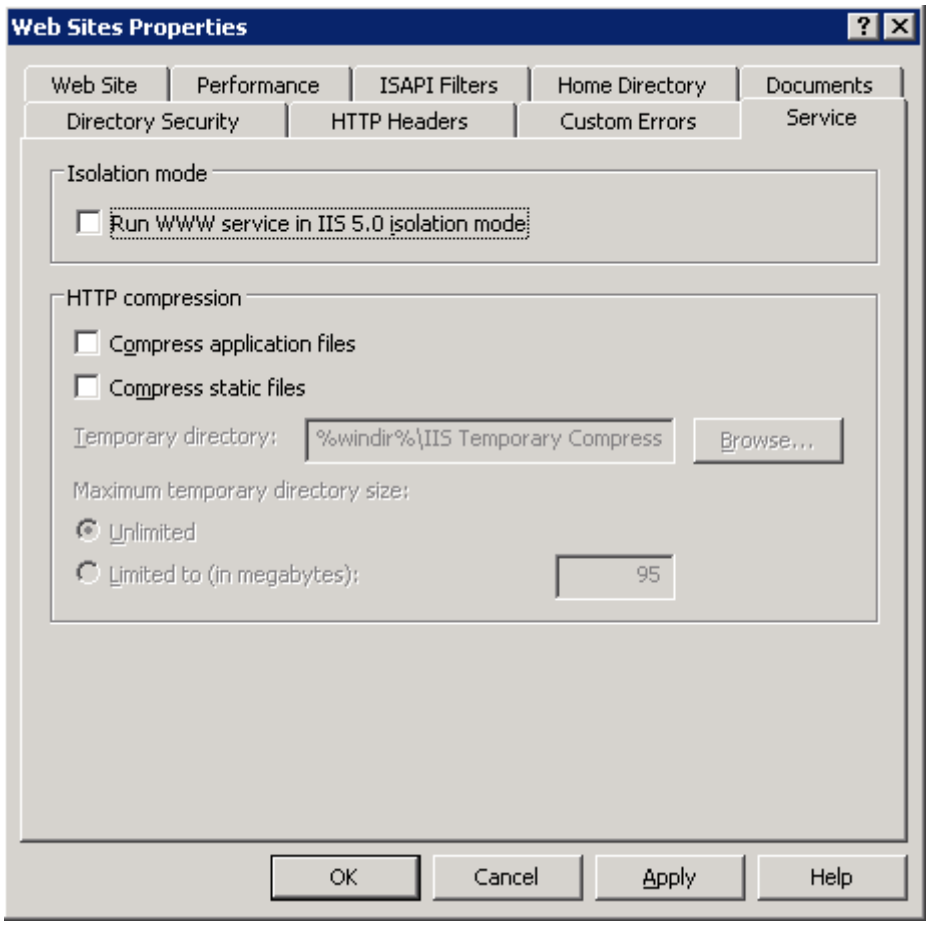
1. Стартираме административната конзола на IIS от Control Panel -> Administrative Tools -> Internet Information Services (IIS) Manager:



2. Избираме с десен бутон на мишката Web Sites и от контекстното меню – Properties:



3. Появява се диалоговия прозорец "Web Sites Properties". Избираме таба Service:



4. В зависимост дали е избрана опцията (Run WWW service in IIS 5.0 isolation mode) се определя изолационния режим, в който се изпълнява Internet Information Server.

5. След промяна на изолационния режим е необходимо рестартиране на IIS, за да влязат в сила направените промени.

Конфигурация на ASP.NET приложенията

Конфигурацията за ASP.NET приложенията се съхраняват в **<processModel>** елемента от файла **machine.config**. Ако се използва IIS 6.0 в режим на изолация на работния процес, се използват само следните настройки:

- **maxWorkerThreads** – максимален брой нишки на процесор, които изпълняват ASP.NET приложенията
- **maxIoThreads** – максимален брой нишки на процесор, които изпълняват входно-изходни задачи

- **responseDeadlockInterval** – задава максимално време, за което всяка заявка трябва да приключи (процесът се рестартира, ако някоя заявка се забави повече)

Всички останали настройки се игнорират. В някои случаи другите настройки са без значение за IIS 6.0, но в други следва техните еквиваленти в IIS 6.0 метабазата да бъдат указани. За повече информация виж "Mapping ASP.NET Process Model Settings to IIS 6.0 Application Pool Settings" в MSDN Library.

Промяна на регистрите на Windows

.NET приложенията би следвало да използват в по-малка степен регистрите на Windows отколкото Win32 приложенията. Например асемблитата не изискват съществуването на ключове в регистрите за разлика от COM базираните обекти в миналото. В някои случаи приложенията все още разчитат на регистрите, например когато:

- Приложенията включват компоненти, които не са изградени с .NET Framework като COM, COM+ или услуги под Windows (Windows services).
- Асемблитата трябва да комуникират с COM базирани компоненти или предоставят услуги, достъпни през COM. В такъв случай записите в регистрите трябва да се направят по време на инсталационния процес.
- Необходимо е добавянето на записи относно лицензиране или контрол на версията.

Споделени инсталационни компоненти (Merge Modules)

Споделените инсталационни компоненти са преизползваеми модули за Windows Installer (обикновено са файлове с разширение **.msm**). Те не могат да се инсталират директно, а трябва да се включат в инсталационния пакет на приложението, което ги използва. Както динамично свързаните библиотеки позволяват преизползването на код и ресурси от няколко приложения, така споделените инсталационни компоненти позволяват споделяне на инсталационен код между MSI пакетите. По този начин се осигуряват едни и същи действия за коректно инсталиране на определен компонент с всички приложения, които го използват. Споделените инсталационни компоненти са подходящи за инсталирани на компоненти на други производители, използвани в нашето приложение (например MSDE – Microsoft SQL Server 2000 Desktop Engine).

Споделените инсталационни компоненти могат да бъдат използвани само в инсталационни пакети на Windows Installer и по никакъв друг начин.

CAB файлове

CAB файлове се използват, за да се пакетират заедно файлове, които са необходими за дадено приложение, така че да могат да се разпространяват по-лесно.

Създаването на CAB файлове предоставя следните предимства:

- Позволяват сваляне от Интернет, а също така контролите с управляван код, които се съдържат в тях, могат да се изпълняват при поискване (on demand).
- Позволяват няколко нива на компресия, което намалява времето за изтеглянето им по мрежата.
- Позволяват използването на Microsoft Authenticode® технологията за подписване на CAB файловете, така че може да се покаже на потребителите кой е производителят.
- Съдържат контроли, които лесно могат да бъдат подменени с по-нова версия, чрез създаване на нов CAB файл и поставянето му на уеб сървър. CAB файловете поддържат версионизиране, като по този начин се осигурява използването на най-новата версия от потребителите.

Локализиране

CLR предоставя поддръжка за извличане на културно-зависими ресурси, които са пакетирани и инсталирани в сателитни асемблита. Сателитните асемблита съдържат само ресурси, които се използват от приложението (**`.resx`**, **`.gif`**, **`.jpeg`** и др.). Те не съдържат изпълним код.

Когато се използва моделът със сателитните асемблита за локализиране на приложенията, структурата е следната: съществува главно асембли съдържащо културата, която се използва по подразбиране и множество сателитни асемблита. В главното асембли се пакетират културно-независимите ресурси и се създава сателитно асембли за всяка култура, която ще се поддържа. Понеже сателитните асемблита не са част от главното асембли тяхната замяна или ъпгрейд са много лесни и се осъществяват без да се променя главното асембли на приложението.

Ако главното асембли на приложението е силно именувано, сателитните асемблита трябва да са силно именувани и подписани със същия частен ключ. В противен случай ресурсите в сателитните асемблита няма да бъдат заредени.

При създаване на .NET приложения, поддържащи множество култури, процесът по разпространение може да се изпълни по няколко начина:

Подход 1: Language packs

Най-лесният подход при разпространение на локализирани приложения е създаване на основен инсталационен пакет и набор от инсталационни пакети за поддържаните култури (познати като езикови пакети - language packs). По този начин клиентите инсталират основния пакет и след това необходимите езикови пакети.

Подход 2: Използване на споделен инсталационен компонент

- Пакетиране на основните, културно-независими асемблита в споделен инсталационен компонент (Merge module) чрез Visual Studio .NET setup and deployment project.
- Създаване на инсталационен проект, който съдържа локализираните ресурси за **всички** поддържани култури.
- Създаване на MSI пакет за всяка култура. Това позволява локализиране и на инсталационния процес, а не само на приложението.
- Създаване на базов инсталационен пакет за основната култура, която ще се поддържа и след това се създава трансформация (transform) за всяка допълнителна култура, която ще се поддържа.
- Добавяне на компилираните асемблита от проекта (project output), който съдържа локализираните ресурси във всеки проект на Windows Installer. Важно е включването на **всички** локализирани ресурси. След това се използва филтър (ExcludeFilter) за изключване всички (освен една) култури от инсталацията.

Подход 3: Инсталиране на всички ресурси

Като друга алтернатива на филтрирането на излишните ресурси по време на създаване на инсталационния пакет е разпространение на всички локализирани ресурси. Единствения недостатък е, че инсталационният пакет може да стане прекалено голям поради излишните сателитни асемблита. Голямото предимство на този подход е, че може да се използват различните култури на една и съща машина. Всичко, което трябва да направи потребителя, е да смени регионалните си настройки и при следващо стартиране на приложението ще се заредят подходящите сателитни асемблита.

Debug Symbols

Когато се създава приложение с помощта на Visual Studio .NET 2003 се създават два вида конфигурации по подразбиране – **Release** и **Debug**. Едно от основните различия между тях е начинът, по който се управляват дебъг символите. С Debug конфигурацията се създава файл с дебъг символите за програмата (**.pdb** – program database), докато при Release такъв файл не се създава по подразбиране.

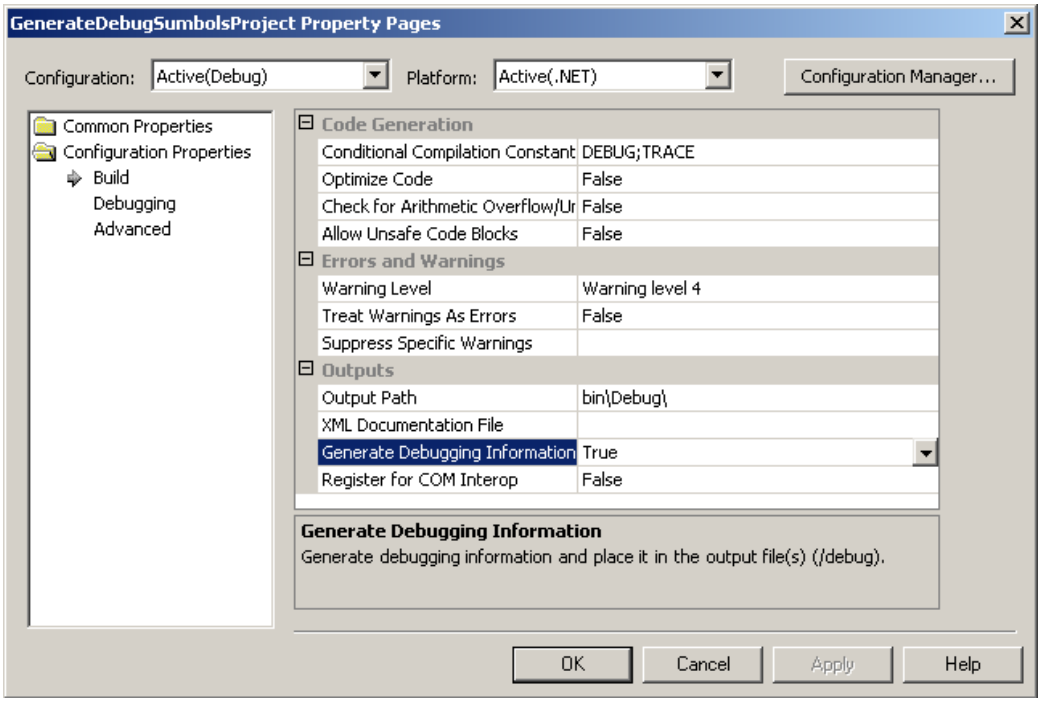
Файлът с дебъг символите е необходим на CLR, за да свърже компилирания MSIL код със сорс кода. Това позволява на дебъг инструментите да показват информация като имена на променливи; на JIT компилатора да създаде проследяваща информация, чрез която да се свърже машинно-зависимия код (native code) обратно до MSIL. Проследяващата информация и символните файлове са необходими за ефективно дебъгване на управлявания код.

Включването на символните файлове в инсталационните пакети помага да се локализират грешките (JIT компилаторът показва дори номера на реда в кода, където е възникнала грешката), затова е препоръчително да се включат в инсталационните пакети по време на тестовете, а при приключване на тестването да се премахнат.

Въпреки, че е удобно намирането на грешки с помощта на символните файлове, те не трябва да се разпространяват заедно с приложението, защото позволяват много лесно да се направи reverse engineering и да се извлече изходния код в почти оригинален вид.

Конфигуриране генерирането на дебъг информация:

Генерирането на символните файлове се определя с **`/debug`** ключа на компилатора от командния ред (**`csc.exe`**) или във Visual Studio .NET от прозореца с характеристиките на проекта (от страницата Build от групата Configuration Properties):



Инсталационни стратегии

Съществуват различни инсталационни стратегии и изборът на една от тях е тясно свързан със спецификата на приложението, на организацията като цяло и бъдещите планове относно приложението. Трябва внимателно да се планира целият процес по разпространение на приложението преди да се предприемат стъпки по изграждане на инсталационните пакети, защото изборът на инсталационна стратегия може изцяло да промени начина, по който се изграждат инсталационните пакети.

Един от основните фактори, които определят стратегията, е типът на приложението – Windows Forms базирано или Web Forms базирано.

Съществуват три основни начина за разпространение на .NET приложения:

- No-touch deployment
- Инсталационни пакети на Windows Installer
- Копиране на файлове

Нека разгледаме подробно всеки един от тях.

No-Touch Deployment (.NET Zero Deployment)

При този подход Windows базирани приложения се поставят на уеб сървър и клиентите ги инсталират като се свързват със сървъра посредством HTTP протокола. При първоначално свързване на клиент се изтеглят асемблитата, които са необходими за първоначална инсталация. След това при първо използване се свалят и реферираните асемблита (on demand). По този начин клиентът не чака да се свалят асемблита, които няма да ползва веднага и така се разпределя мрежовото натоварване. Асемблитата се свалят в Assembly Download Cache (**<windir>\assembly\download**) и се съхраняват там.

Изключителното предимство на тази технология е, че се комбинира богатият потребителски интерфейс на Windows базираните приложения с лесната инсталация и поддръжка, характерна за уеб приложенията. Понеже асемблитата се свалят само когато са необходими, се минимизира времето за начално зареждане на приложението. Всичко това се случва автоматично – когато клас от главното асембли създава инстанция на клас от асембли, което се намира в същата папка на уеб сървъра, CLR го сваля.

При всяко стартиране на приложението (и първоначално използване на асембли) CLR проверява дали асемблитата на уеб сървъра имат по-нови версии от тези в кеша и при необходимост сваля по-новите версии. По този начин инсталирането на по-нови версии е изключително лесно, като всичко, което е необходимо да се направи, е да се заменят асемблитата на уеб сървъра с по-нови версии.

Кога да ползваме No-Touch Deployment?

В някои случаи използването на no-touch deployment не е подходящо:

- При необходимост да се контролира строго мрежовият трафик и да се прогнозира неговото използване. Тъй като асемблитата се свалят само когато са необходими, съществува потенциална възможност за пикови натоварвания на мрежовите ресурси, когато много потребители изискват едновременно някои асемблита.
- При необходимост от използване на приложението offline. Съхраняване на асемблитата в кеша позволява тяхното стартиране дори когато няма връзка с уеб сървъра, но в такъв случай ще се използват само вече кешираните асемблита. При използване на функционалност от асемблита, които не са свалени, ще възникне грешка, тъй като CLR не може да ги изтегли.
- Когато се изискват допълнителни действия по време на инсталационния процес като инсталиране на COM обект или драйвер за устройство.
- Когато са необходими високи права. По подразбиране приложенията, разпространявани с no-touch deployment, се стартират в ограничен контекст на сигурността. Не е подходящо използването на тази технология, когато не е практично да се променя политиката по сигурността (security policy).
- Когато е необходимо поставянето на асембли в Global Assembly Cache.

Дали приложението работи в online или offline режим се определя от режима на Internet Explorer. Дори и компютърът да има връзка с даден уеб сървър, ако Internet Explorer е в offline режим, тогава и CLR ще работи в offline режим.

Ако смятате да разпространявате приложението чрез .NET Zero Deployment, е добре планирането да започне още от етапа на дизайна му. Това ще позволи решаването на някои проблеми при дизайна (като например ограничените права на приложението) вместо срещането им след като приложението е вече в употреба.

Възможно е използването на устройство в локалната мрежа (network share), вместо уеб сървър. По този начин потребителите се свързват и директно стартират приложението оттам. Чрез този подход асемблитата не се кешират, а се зареждат директно в паметта. Основен недостатък е, че е невъзможно offline използването при срив на мрежовото устройство.

Споделени асемблита

.NET Zero Deployment не поддържа инсталиране на споделени асемблита в Global Assembly Cache. Това означава, че ако е необходимо да се поставят определени асемблита там, трябва да се използва друг инсталационен

механизъм. Преди да се възприеме тази техника трябва внимателно да се реши трябва ли да се инсталират асемблита в Global Assembly Cache въобще. Трябва да се има предвид алтернативата да се разпространяват асемблитата като частни (виж Частни асемблита) с всяко приложение. Частните асемблита създават проблеми с ъпгрейда на асемблитата, но това се компенсира от лекотата, с която се разпространяват нови версии при .NET Zero Deployment (просто копиране на уеб сървъра и при следващото стартиране всеки клиент ще използва новата версия).

Приложения, състоящи се от множество асемблита

Приложения, които се състоят от едно асембли са най-лесни за разпространение чрез .NET Zero Deployment. Често, обаче, напълно функционалните приложения използват множество асемблита и множество ресурси. Разпространението на такива приложения с .NET Zero Deployment е възможно, но изисква допълнителни усилия от разработчиците, за да се осигури оптимизация.

С цел извикване на асемблита само когато са необходими може да се използва класът **`Assembly`** от пространството **`System.Reflection.Assembly`**. В този клас има метод **`LoadFrom(string path, …)`**, чийто параметър **`path`** приема както URL, така и пълно име на локален файл. Когато е подаден URL като параметър, CLR проверява дали изискваното асембли не съществува вече в download кеша. Предимството на тази техника е, че асемблито се сваля от уеб сървъра, само когато е извикано от приложението (on demand), въпреки, че може да доведе до забавяне на приложението докато асемблито се свали на локалната машина.

Решение на проблема с ограниченията на .NET Zero Deployment технологията е използване на комбиниран метод на инсталация – компонентите, за които трябва да се предоставят често нови версии, се поставят на уеб сървър, а политиката за сигурността и останалите компоненти се инсталират чрез Windows Installer технологията.

Този подход набра популярност и Майкрософт решиха да го доразвият във .NET Framework 2.0 и нарекоха технологията ClickOnce. Във Visual Studio .NET 2005 и .NET Framework 2.0 основните възможности, които са добавени, са следните:

- Уведомяване на клиентите при публикуване на нова версия.
- Избор на потребителите дали да инсталират новата версия - разработчиците могат да посочат най-старата версия, която е допустимо да бъде стартирана.
- Инсталиране/деинсталиране – създаване/изтриване на препратки (shortcuts) в подходящите папки.
- Генериране на подходящи уеб страници за уведомяване на потребителя.

- Опция за стартиране в offline режим – улеснено е в сравнение с .NET Framework 1.x.

Windows Installer

Следващата стратегия за разпространения на приложенията е чрез инсталационни пакети на Windows Installer (**.msi** файлове). Тази стратегия предлага най-много възможности сред изброените и чрез нея могат да се инсталират всички видове .NET приложения заедно със споделените инсталационни компоненти (merge modules) и **CAB** файлове.

Предимства на Windows Installer

Windows Installer предлага множество улеснения за потребителите, администраторите и разработчиците. Основните от тях са:

- Лесен за използване потребителски интерфейс, който може да се настройва от разработчиците (customizable UI).
- Интеграция с инструмента Add/Remove Programs от Control Panel за следните действия:
  - Инсталиране.
  - Деинсталиране.
  - Добавяне или премахване на функционалност (features) от приложенията.
  - Поправяне на инсталираното приложение като се запазват направените промени и се възстановяват повредените файлове.
- Изпълнение в тих режим (silent mode) – без намесата на потребителя.
- Възстановяване на системата до състоянието преди започване на инсталацията в случай на:
  - Възникване на грешка.
  - Прекъсване от потребителя.
- Проверяване за наличието на определен софтуер и хардуер преди започване на инсталацията.
- Създаване на подходящи препратки (shortcuts).
- Управление на местоположението на файловете и папките.
- Управление на регистрите на Windows (Windows registry).
- Инсталиране на COM базирани компоненти.
- Инсталиране на асемблита в Global Assembly Cache.
- Изпълнение на допълнителни действия след инсталацията (custom actions).

- Управление на информацията за версиите, като по този начин се осигурява инсталиране на надстройките (upgrades) и кръпките (patches) в правилен ред.

Както се вижда от посочения списък с предимствата на използването на Windows Installer, това е много мощна технология и използването ѝ позволява решаване на проблемите, съпътстващи сложните Windows и уеб базирани приложения. Ето и някои от тях:

Задача	Решение чрез MSI пакет
Споделени асемблита – поставяне на асемблита в GAC	Windows Installer предоставя лесен начин за инсталиране на споделени асемблита в Global Assembly Cache.
Инсталиране на COM базирани компоненти	Както обяснихме в точка COM базирани обекти, COM обектите трябва да се регистрират преди да се използват. Windows Installer предоставя надежден механизъм за инсталиране и регистриране на COM обекти. При нужда може да се използва и инструмента **`RegSvr32.exe`** за ръчно регистриране на COM обектите.
Настройки на IIS	В точка Настройки на Internet Information Server (IIS) обяснихме основните настройки, които трябва да бъдат направени, за да се стартират уеб приложения. Windows Installer позволява задаването на тези настройки по време на създаване на инсталационния пакет.
Ресурси на приложението	Приложенията изискват различни ресурси като опашки от съобщения (message queues), логове на събитията (event logs), индикатори за производителността (performance counters) и бази данни (databases) (виж Инсталационни компоненти), сателитни асемблита за локализиране на приложението (виж. Локализиране). Windows Installer поддържа изпълнението на допълнителни действия (custom actions) преди приключване на инсталационния процес, чрез които могат да се изпълнят почти всички необходими действия.

Windows Installer за инсталиране на многослойни приложения

При комплексни системи, изградени от няколко слоя, трябва да се използва отделен инсталационен пакет за всеки отделен физически слой (всяка отделна машина). Това е подходящо при повечето сценарии, защото е изключително трудно (в някои случаи дори невъзможно) да се стартира един инсталационен процес на даден физически слой и да се инсталират

компоненти на други физически слоеве като част от него. Например ако се стартира инсталационен пакет за уеб приложение, е трудно да се инсталират компонентите, които съдържат бизнес логиката, на отделен компютър. В някои случаи за всеки отделен физически слой може да бъде избрана различна инсталационна стратегия. Например слоят, който съдържа уеб-базираният потребителски интерфейс може да бъде разпространен като колекция от файлове след компилацията, докато слоят с бизнес логиката и сървърът за базата данни могат да бъдат поставени в MSI пакет.

Съображения за сигурността

Един от проблемите, които засягат разпространението на Windows базирани приложения чрез MSI пакети е дали потребителят, който е стартирал инсталационния процес, има необходимите права, за да проведе и приключи инсталацията. Тези права зависят от действията, които се извършват по време на инсталационния процес и платформата, върху която се осъществяват. Например не се изискват специални права, за да се инсталират приложения върху Windows 95, Windows 98, Windows Me, докато в Windows NT/2000/XP/2003 само създаването на поддиректория на системната папка **`Program Files`** изисква потребителят да е в група със специални права (като **`Administrators`** или **`Power Users`**).

Начин да се подсигурим, че инсталацията няма да се провали заради недостатъчни права на активния потребител, е разпространение на инсталационния пакет чрез MS SMS или Active Directory Group Policy.

Разпространение на пакети на Windows Installer

Съществуват различни начини за разпространение на MSI пакети:

- Active Directory Group Policy.
- MS System Management Server (SMS).
- Други начини – поставяне на файлов сървър, на уеб сървър или разпространение на носител (CD/DVD).

Нека разгледаме всеки един от тях.

Групови политики на активните директории (Active Directory Group Policy)

При работа в големи организации работните станции се организират в домейни, управлявани от т. нар. активна директория (Active Directory). Тя е част от сървърните Windows платформи (Windows 2000 Server, Windows 2003 Server и т. н.) и се използва за централизирано управление на Windows базирани корпоративни инфраструктури.

Активната директория позволява разпространението на приложение до потребителите или машините автоматично, чрез използването на груповите политики (Group Policy). Груповите политики могат да се задават на ниво домейн, организационна единица, потребител или компютър. Това

зависи от структурата на дадената организация. Чрез груповите политики може да се осигури автоматично инсталиране на приложението, когато даден потребител се включи в системата (или даден компютър се стартира).

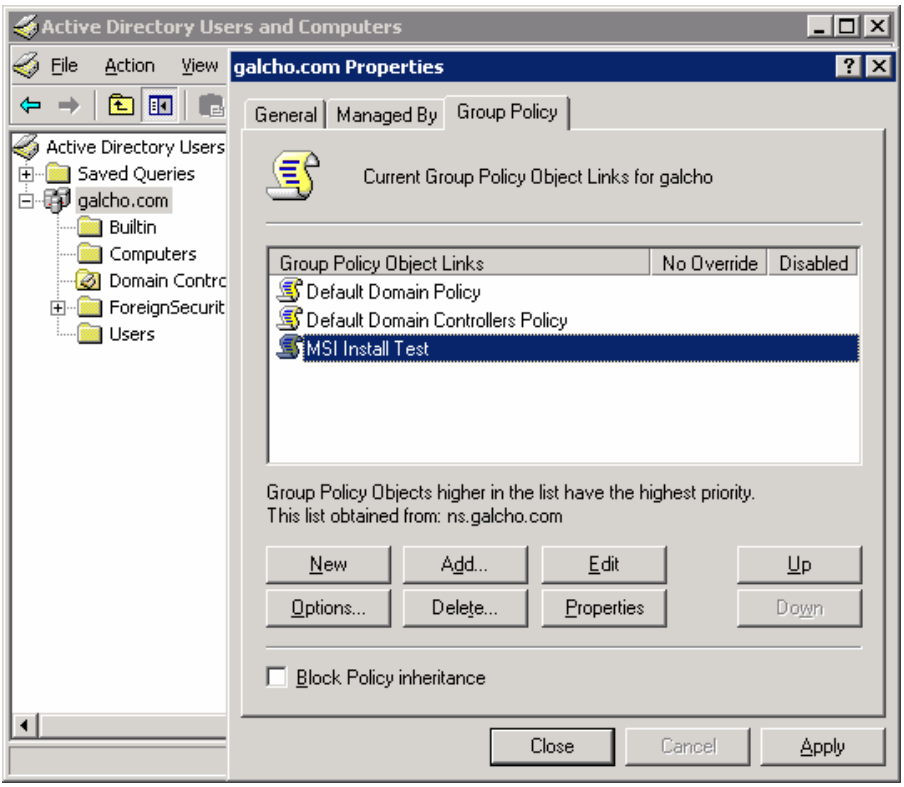
Груповите политики позволяват разпространение на приложения по два начина:

- Назначение (assign) – администраторът може да назначи дадено приложение за потребители или машини.
    - За потребители – приложението се инсталира, когато даденият потребител се включи в системата. Когато потребителят стартира някоя програма за първи път, тогава инсталацията се финализира.
    - За машини – когато машината се стартира, приложението се инсталира и то е свободно за използване от всички потребители на тази машина. Инсталацията се финализира, когато потребителят стартира някоя програма.
- Публикуване (publishing) – приложението може да се публикува за определени потребители. Когато те се включат публикуваната програма се появява в Add/Remove Programs и може да бъде инсталирана от там. Като алтернатива може да се посочи приложението да се инсталира при стартиране на файл, чийто тип е асоцииран с него.

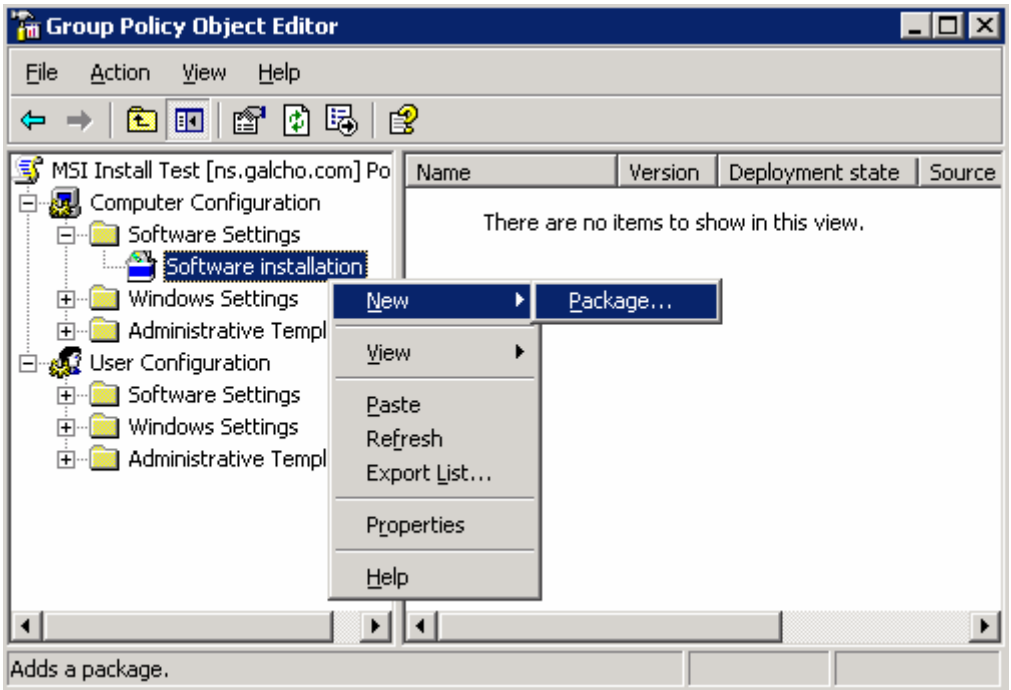
Разпространение на MSI пакет чрез групова политика на активната директория – пример

За да настроим инсталационен пакет за инсталиране чрез груповата политика на активната директория (Active Directory Group Policy), можем да изпълним следните стъпки (примерът е с Windows 2003 Server):

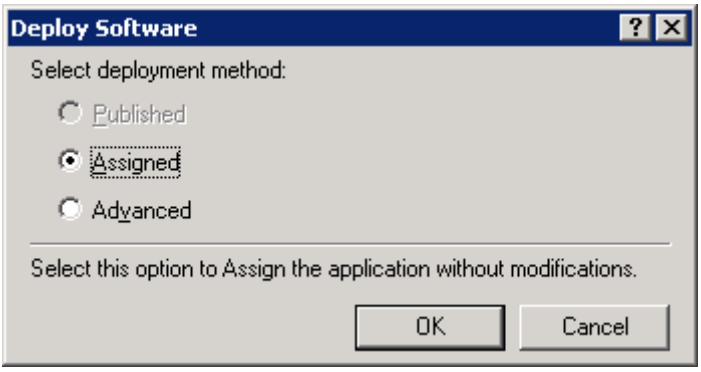
1. Създаваме директория, която ще съдържа MSI пакета на файлов сървър. Настройваме директория за съвместно ползване (shared directory) и задаваме необходимите права.
2. Стартираме конзолата "Active Directory Users and Computers".
3. От структурата вляво избираме контейнера, който съдържа компютрите, за които ще назначаваме инсталация на приложение. Щракваме с десния бутон на мишката върху него и избираме Properties и после Group Policy таба.
4. Създаваме нова групова политика (Group Policy Object), чрез бутона **[Add]** и задаваме подходящо име. Например "MSI Install Test":



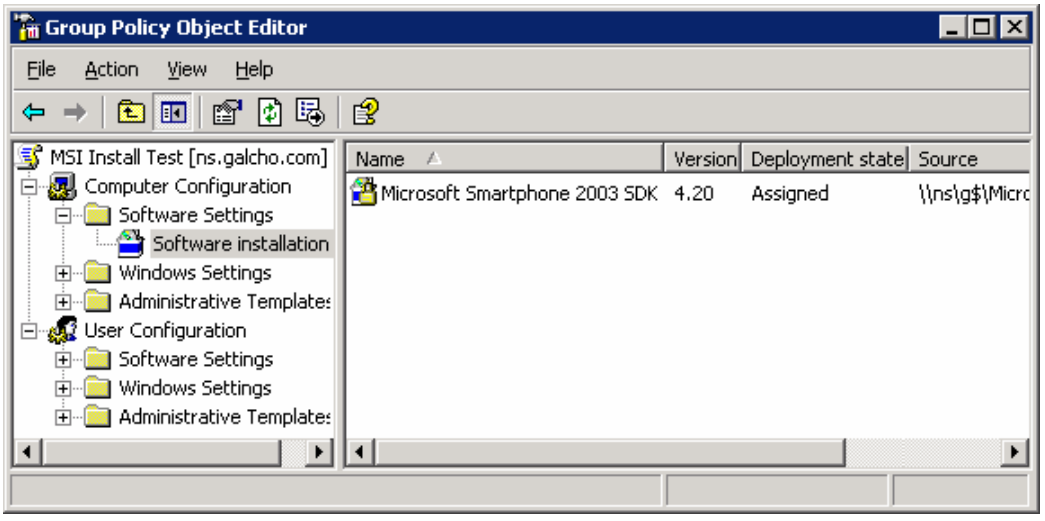
5. Уверяваме се, че новата група е избрана и натискаме бутона **`[Edit]`**. Ще се отвори конзолата Group Policy Object Editor.



6. От дървото вляво отваряме Computer Configuration, разпъваме папката Software Settings и избираме иконата Software Installation. От нейното контекстно меню избираме New… | Package.
7. Отваря се диалог, чрез който трябва да изберем `.msi` файла за този пакет. Намираме споделената папка на файловия сървър, която създадохме в стъпка 1. Избираме файла и потвърждаваме с бутона `[Open]`. Ако файлът се намира на локалния диск, не трябва да използваме локалния път (примерно `c:\PathToMSI`), защото клиентите няма да имат достъп до пакета. Вместо това трябва да използваме UNC път – \\име-на-сървъра\име-на-папката\файл.msi.
8. От следващия диалогов прозорец избираме `[Assigned]` и потвърждаваме с `[OK]`:



Ако изберем вместо това `[Advanced]`, ще се отвори нов прозорец, в който можем да правим допълнителни настройки за разпространението на пакета. Това може да стане и по-късно чрез избиране на Properties от контекстното меню на създадения пакет.



9. Виждаме новосъздадения пакет в конзолата Group Policy Object Editor. В нашия случай той ще бъде инсталиран при следващото

влизане на потребителите в системата или при стартиране на компютрите, които са част от домейна.

MS System Management Server (SMS)

System Management Server v2.0 служи за лесно централизирано управление на инсталации на софтуерни пакети в корпоративна мрежа. Основните предимства на SMS са:

- Разпространение на приложения на множество клиентски платформи. Всички версии на Windows се поддържат от SMS.
- Контрол над натоварването на мрежата. SMS позволява наблюдение на мрежовия канал и неговото натоварване като позволява настройване и избягва допълнителното натоварване в неподходящо време от денонощието.
- Работа с определени потребители и машини. Приложенията може да се разпространяват на база потребителско име, име на група, име на компютър, име на домейн и мрежов адрес.
- Използване на график за разпространение на софтуер. Разпространението се извършва по предварително указан график. Това е полезно за избягване на натоварването по определено време на денонощието.
- Състояние на разпространението. SMS показва състоянието на инсталацията и позволява навременно реагиране в случай на грешки.
- Краен резултат на разпространението. Ако приложението е било успешно инсталирано на даден клиент, това не означава, че целият процес е бил успешен. SMS предоставя детайлна информация за крайния резултат.

SMS има свой механизъм за разпространение на приложенията и не изисква задължителното използване на пакети на Windows Installer, но поддържа MSI пакети.

Други методи за разпространение на MSI пакети

Съществуват и няколко други метода за разпространение на MSI пакети:

Метод	Описание	Предимства/Недостатъци
Уеб/FTP сървър	Пакетите са поставени на сървър от локалната мрежа или в Интернет и връзка към тях е изпратена на	Добър начин за предоставяне на приложение на широка аудитория. Пакетите могат да бъдат архивирани (например в .ZIP архив), за да се избегне директно стартиране. Няма контрол върху свалянето на пакетите. Изискват се административни права за ус-

	потребителите.	пешна инсталация.
Мрежов сървър	Пакетите са поставени на сървър и връзка към тях е изпратена на потребителите.	Добър начин за разпространение в дадена организация. Пакетите могат да бъдат архивирани (например в .ZIP архив), за да се избегне директно стартиране. Няма контрол върху свалянето на пакетите. Изискват се административни права за успешна инсталация.
E-mail	Пакетите се изпращат чрез електронната поща	Лесни са за намиране. Не могат да се стартират директно, заради блокирането им от клиентите за електронна поща. Могат да бъдат филтрирани от системи за защита от спам и вируси. Може да доведе до значително натоварване на сървърите. Изискват се административни права за успешна инсталация.
CD/DVD	Пакетите са записани на оптичен носител.	Лесно преносими. Подходящи, където мрежовият канал не е достатъчно широк или пакетите са много обемни. Изискват се административни права за успешна инсталация.

Инструменти за създаване на MSI пакети

Visual Studio .NET 2003 поддържа създаването на инсталационни пакети, въпреки че не използва пълните възможности на технологията Windows Installer. Затова ще посочим най-често използваните инструменти за работа с MSI пакети. Основно може да разделим инструментите на две групи:

1. Доставяни от Майкрософт:

   - Visual Studio .NET 2003 – поддържа създаването на основни инсталационни пакети и предлага сравнително добър интерфейс за изграждане на инсталационния процес. Подходящ е за средни по сложност приложения.
   - Orca – съдържа се в Platform SDK. Предлага среда за редактиране на файлове на Windows Installer (**`.msp`**, **`.msi`**, **`.msm`**). Изключително мощен, но сложен – подходящ за експерти в технологията Windows Installer.

- Windows Installer XML (WiX) – това е проект по инициатива на Майкрософт Shared Source Licensing (виж http://www.microsoft.com/resources/sharedsource/licensing/WiX.mspx и http://sourceforge.net/projects/wix/). Позволява за описанието на MSI пакетите да се използва XML сорс код, който се компилира до **.msi** файл. Подходящ за напреднали разработчици.

2. От други производители:

- InstallShield – продукт на Macrovision (бившата InstallShield). Изключително интуитивен и лесен за използване интерфейс. Работата е улеснена от наличието на съветници (wizards). Интегрира се отлично във Visual Studio .NET 2003. Документацията е много подробна и за най-често използваните задачи са дадени примери. Предоставя свой собствен скриптов език (InstallScript), който позволява контролиране на инсталационния процес във всичките му аспекти. Официалният уеб сайт на InstallShield е http://www.installshield.com/products/installshield/.
- Wise for Windows Installer – продукт на Wise, който се интегрира с Visual Studio .NET 2003. Мощен продукт и лесен за употреба. Също предоставя свой скриптов език за контролиране на инсталационния процес. Официалният уеб сайт на Wise for Windows Installer е http://www.wise.com/wfwi.asp.

Колекция от файлове след компилация

За много уеб приложения и някои опростени Windows приложения е по-подходящо разпространението чрез просто копиране на файлове на сървъра, вместо изграждането на сложни MSI пакети. Под колекция от файлове имаме предвид всички файлове, които се използват от приложението – **.apsx**, **.dll**, **.exe**, **.config**, графични файлове и други ресурси.

Предимствата на тази инсталационна стратегия са:

- Леснота за инсталиране – файловете се инсталират на машината чрез просто копиране.
- Леснота за ъпгрейд – новите файлове се копират върху старите.

За разлика от Windows базираните приложения, уеб приложенията се инсталират от администратор или опитен IT специалист. В много случаи такива приложения не се инсталират, деинсталират и поправят чрез Add/Remove Programs от Control Panel. Възстановяване на предишното състояние на машината при грешка също не е необходимо условие – полесно е за администратора да промени настройките на IIS или да отстрани дребни проблеми вместо да рестартира целия инсталационен процес.

Въпреки, че тази стратегия работи добре с по-прости приложения, тя не е подходяща при следните ситуации:

- промяна в Windows Registry

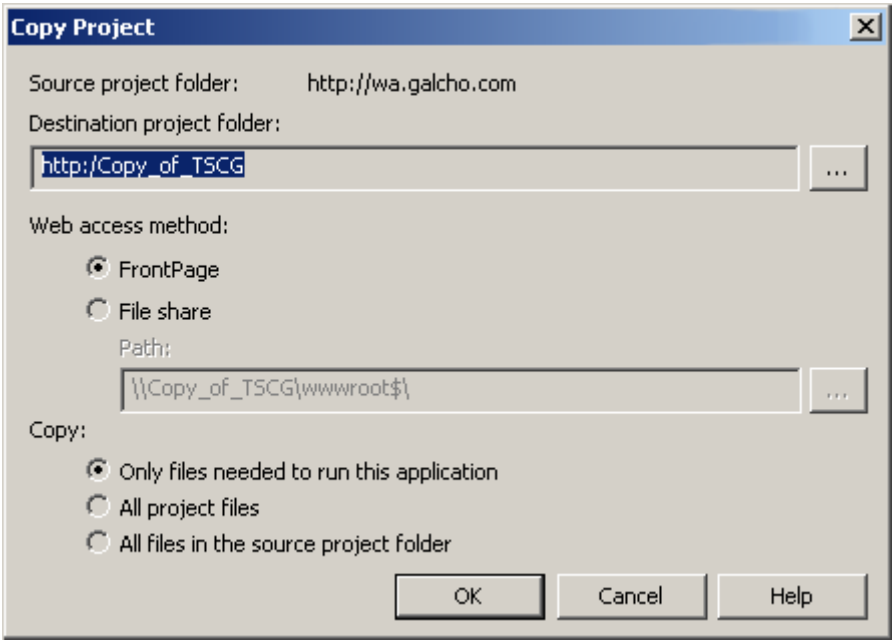
- добавяне, изтриване или промяна на Windows услуги
- промяна по политиките на сигурността (security policy)
- добавяне, изтриване или промяна на COM базирани обекти
- работа с Global Assembly Cache

При работа с по-сложни приложения е препоръчително използването на Windows Installer.

Начини на разпространение

Самото разпространение на файловете може да се извърши по следните начини:

- Чрез Microsoft Application Center – изключително подходящо при т. нар. уеб ферми (Web farms). Те представляват клъстер от няколко машини, които предоставят уеб услуга или уеб приложение. При инсталиране на уеб приложение върху уеб ферма е необходимо всяка една от машините да има едни и същи файлове, инсталирани компоненти, настройки на IIS и др. Използването на Microsoft Application Center осигурява съдържанието на машините в уеб фермата да бъде еднакво. Microsoft Application Center не е подходящ за разпространение на MSI пакети, Windows базирани приложения, Windows услуги и бази данни.
- Чрез Copy Project командата от Visual Studio .NET 2003 (само за уеб приложения). Създава виртуална директория на посочения сървър и копира файловете в нея:



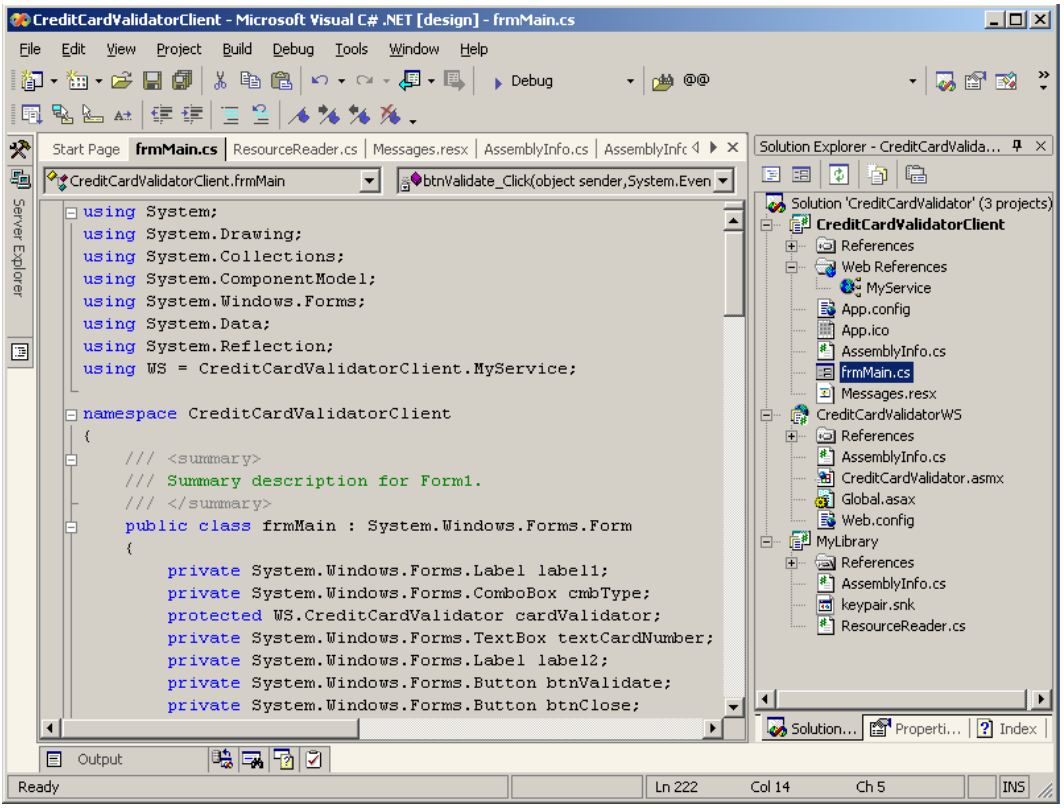
Има следните опции за копиране:

- Само файловете, необходими за изпълнение на приложението – включва: файловете създадени по време на компилация, реферираните асемблита, както и всички файлове, които са добавени в проекта на Visual Studio .NET и за които е зададено **`BuildAction=true`**.
- Всички файлове от предходната точка и всички проектни файлове.
- Всички файлове от проектната директория и всички поддиректории.

- Директно копиране на файлове – подходящо е само, ако не трябва да се променят регистрите на Windows (Windows registry) или да се изпълняват допълнителни задачи (като настройки и рестартиране на IIS, промяна на настройките за сигурността и т.н.).

Създаване на MSI инсталационен пакет

След като описахме различните инсталационни стратегии е време да преминем към практическата част. За съжаление няма да можем да обхванем всички описани варианти за разпространение, но ще демонстрираме най-важните от тях. Ще се спрем на два вида MSI пакети – за Windows базирано приложение и за уеб базирано приложение.



Примерът, който ще разгледаме, включва следните проекти:

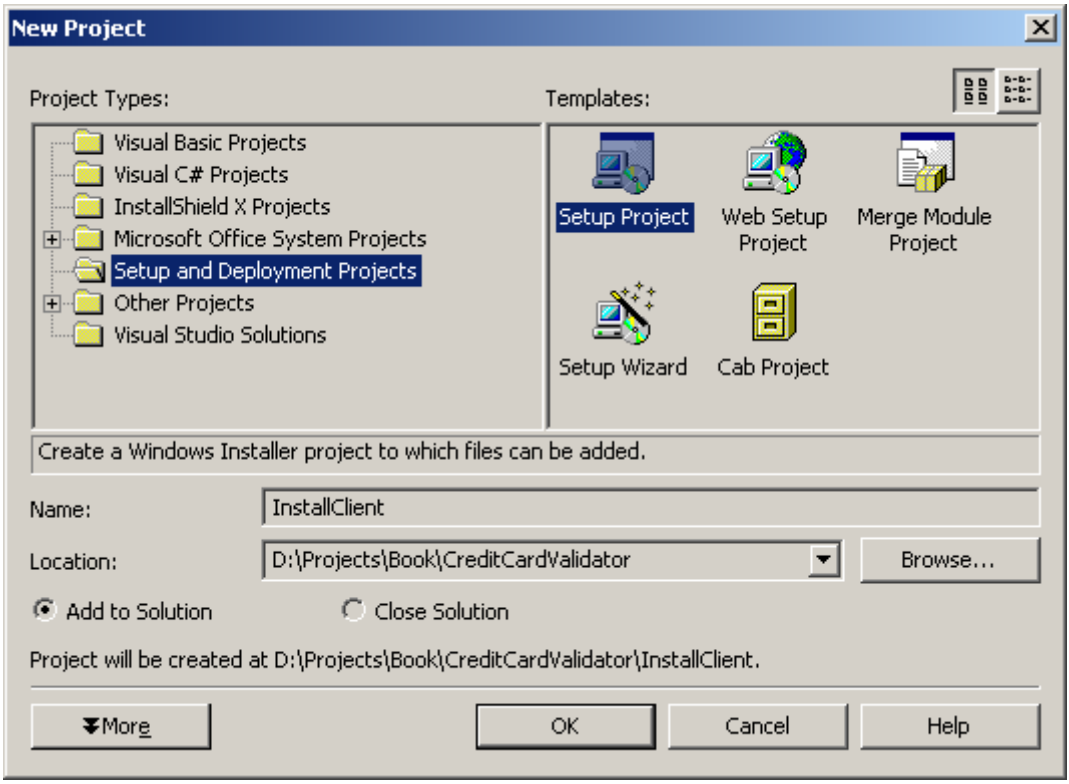
- **`CreditCardValidatorWS`** – уеб услуга, която проверява валидността на кредитна карта.
- **`CreditCardValidatorClient`** – Windows базирано приложение, което използва методите на **`CreditCardValidatorWS`**.
- **`MyLibrary`** – библиотека от класове, съдържаща клас за четене на ресурсни файлове, които са компилирани в асемблитата. Силно именуваното асембли **`MyLibrary.dll`** ще бъде добавено в GAC.

По-горе е показано как изглежда във VS.NET решението, което включва трите проекта.

Няма да навлизаме в описание на тези проекти, понеже целта е друга – да покажем процеса на създаване на инсталационни пакети за всеки от тях.

Създаване на инсталационен пакет на Windows базирано приложение

Нека първо създадем инсталационен пакет за Windows Forms базираното приложение **`CreditCardValidatorClient`**.



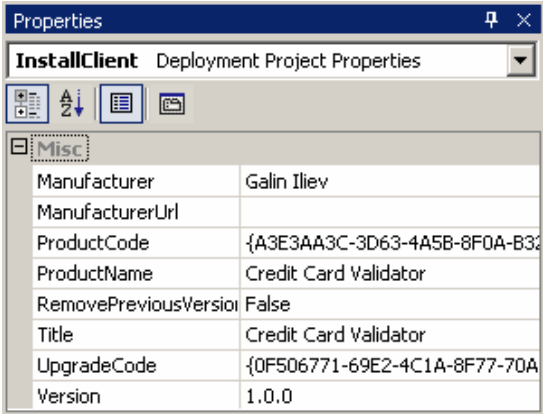
От зареденото решение **`CreditCardValidator.sln`** във Visual Studio .NET 2003, избираме File -> New -> Project и избираме Setup Project от

категорията Setup and Deployment Projects. За име въвеждаме **`InstallClient`** и избираме опцията **`Add to Solution`** (вж. фигурата по-горе). Потвърждаваме с бутона **`[OK]`**.

Новият проект се създава и се добавя към решението **`CreditCardValidator.sln`**.

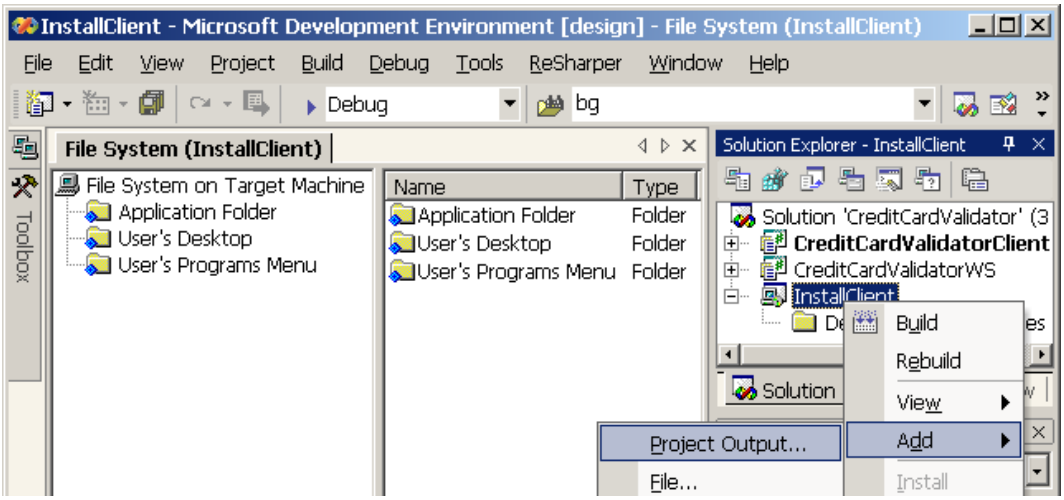
Характеристики на инсталационния проект

Трябва да зададем свойствата на проекта – да зададем име на продукта, код, производител и др. За целта избираме проекта **`InstallClient`** и отваряме прозореца **`Properties`** на Visual Studio .NET 2003. Задаваме стойности на свойствата **`Title`** и **`ProductName`** "Credit Card Validator". Задаваме и автор и производител на продукта, ако е необходимо:

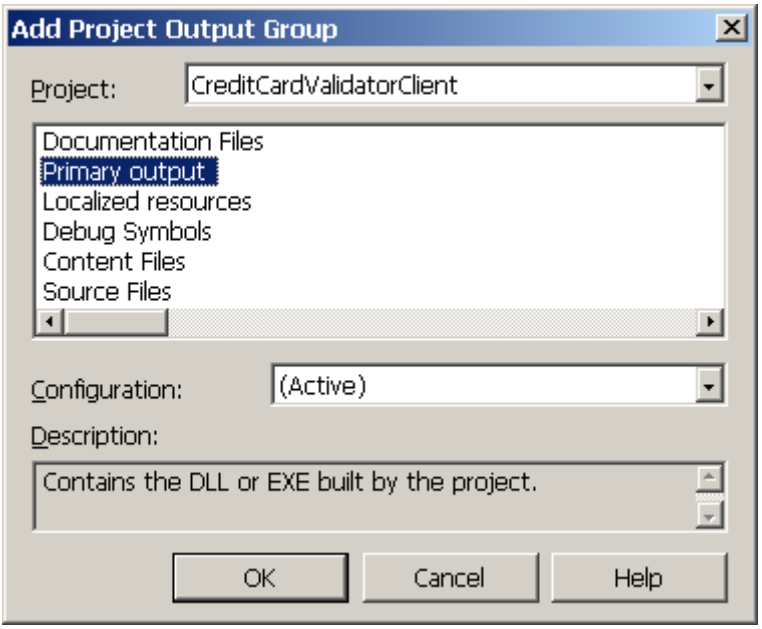


Добавяне на файлове към инсталационния проект

Следващата стъпка е добавяне на проектните файлове към създадения инсталационен проект. От контекстното меню на проекта **`InstallClient`** избираме Add -> Project Output…



Отваря се диалогов прозорец, от който се избира кои части на проекта (и конкретния проект) да се включат в инсталацията. Избираме проекта **`CreditCardValidatorClient`**, избираме **`Primary Output`**. (Тази опция добавя само файловете, които се създават по време на компилация. Всички други файлове трябва да се добавят допълнително.)

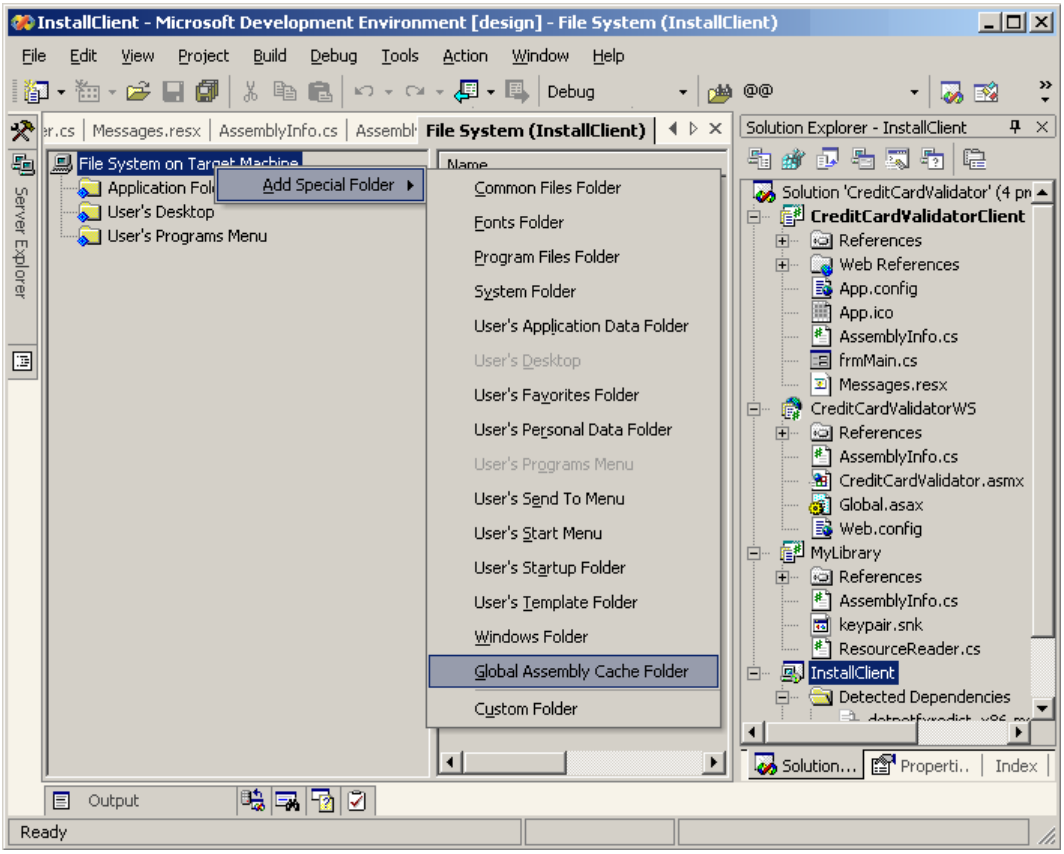


Потвърждаваме с бутона **`[OK]`** и към инсталационния проект е добавен **`Primary Output`**. Обърнете внимание, че Visual Studio .NET 2003 е добавило реферираното асембли **`MyLibrary.dll`** в категорията **`Detected Dependencies`**.

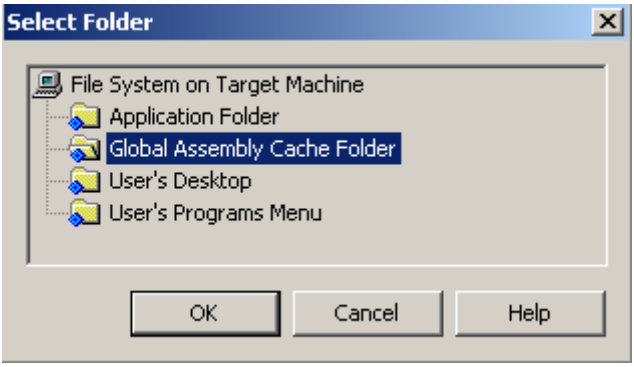
По подразбиране папката, в която ще бъде инсталирано **`MyLibrary.dll`**, е инсталационната папка на приложението, която потребителят избира по време на инсталация.

За да инсталираме **`MyLibrary.dll`** в Global Assembly Cache трябва да направим допълнителни настройки. Ето стъпките, които трябва да извършим:

В изгледа **`File System`** на инсталационния проект (отваря се от контекстното меню на инсталационния проект **`InstallClient`** -> View -> File System) отваряме контекстното меню на File System on Target Machine -> Add Special Folder -> Global Assembly Cache Folder:



След като вече имаме добавен GAC към инсталационните папки на проекта може да зададем местоположението на **`MyLibrary.dll`**. Отваряме контекстното меню на **`MyLibrary.dll`** от Solution Explorer (от категорията Detected Dependencies на проекта **`InstallClient`**) и избираме Properties. От прозореца Properties избираме свойството Folder. Неговата стойност по подразбиране е Application Folder. Избираме бутона ... и от диалоговия прозорец Select Folder избираме Global Assembly Cache и потвърждаваме с бутона **`[OK]`**.

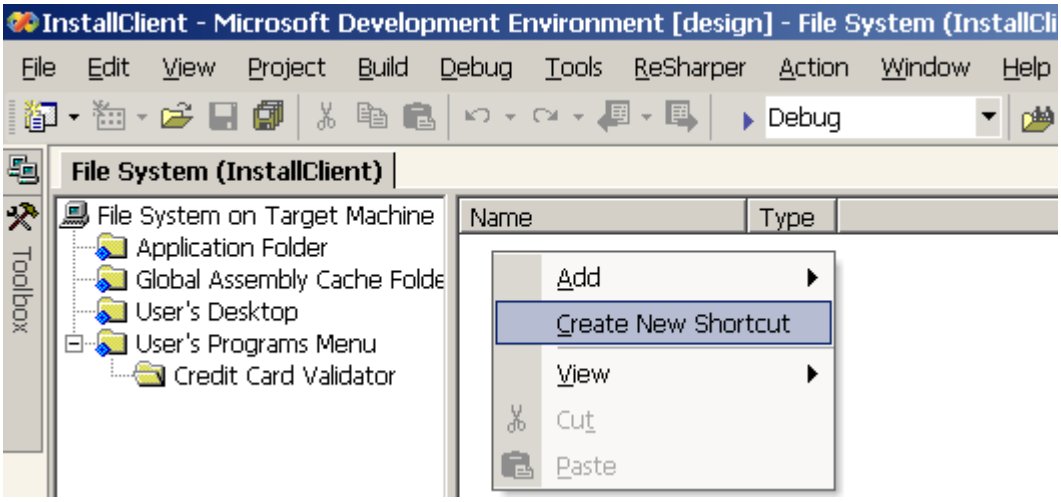


Създаване на икони за инсталираното приложение

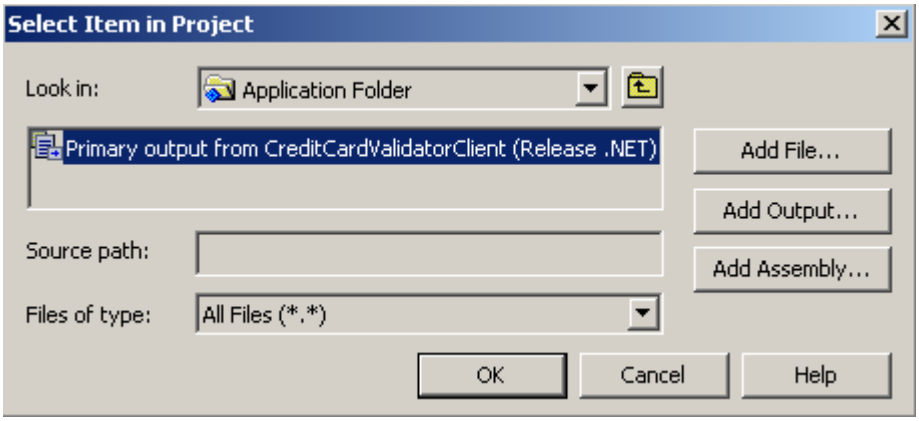
За да е завършен инсталационния процес трябва да добавим препратки (shortcuts) за бързо стартиране на приложението. За програмни продукти, насочени към обикновените потребители (не администратори) е задължително добавянето на препратки (shortcuts) в Start Menu на Windows. Добавянето на препратки (shortcuts) на работната площ (desktop) и лентата за бързо стартиране (Quick Launch) не е задължително, но може да се извърши за удобство на потребителя.

Добавяне на препратки в Start Menu на Windows

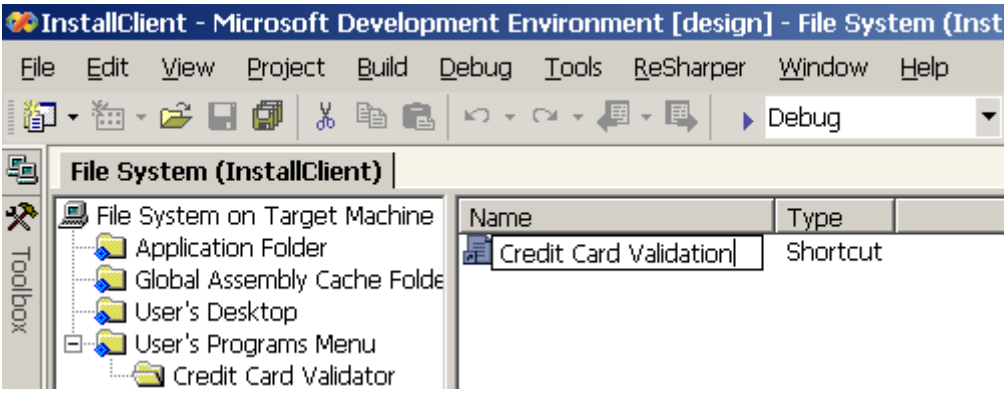
От изгледа **`File System`** избираме **`User's Programs Menu`** с десен бутон на мишката и от контекстното меню избираме Add -> Folder. За име на папката въвеждаме "**`Credit Card Validator`**". Избираме я и от контекстното меню на детайлната област (в средата) избираме **`Create New Shortcut`**.



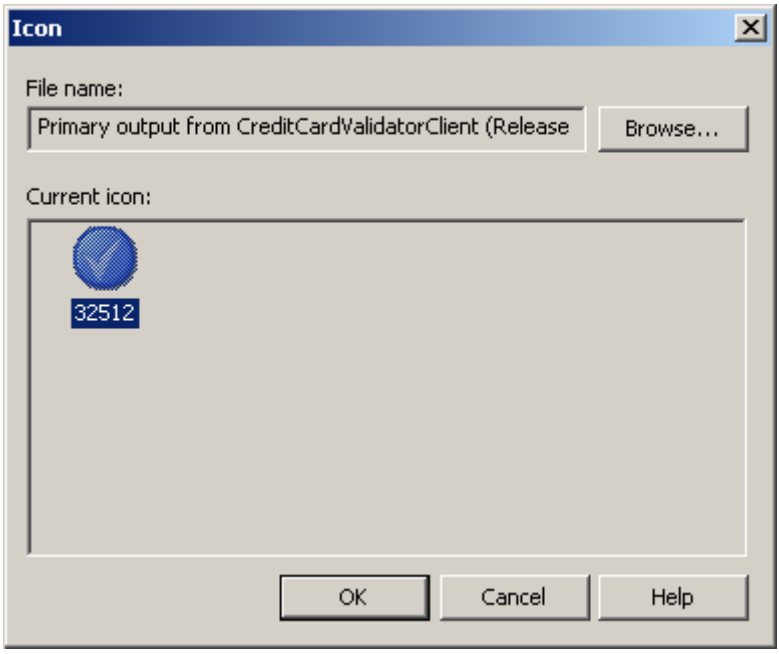
От отворилия се диалогов прозорец избираме Application Folder -> Primary Output for **`CreditCardValidatorClient`** (Release .NET) и потвърждаваме с **`[OK]`** бутона:



В детайлната област се добавя препратка към основния файл на приложението. Името е в режим на редактиране и можем да въведем подходящо име, напр. "**`Credit Card Validation`**":



По подразбиране липсва икона за тази препратка. От прозореца Properties избираме свойството Icon и се отваря прозорец за избор на икона. Файлът трябва да е включен в инсталацията. Добра практика е да се избира иконата от изпълнимия файл, към който сочи препратката. Ако изберем **`.exe`** от проекта **`CreditCardValidatorClient`**, се показват всички икони, които се съдържат в него:



Избираме подходяща икона и потвърждаваме с бутона `[OK]`.

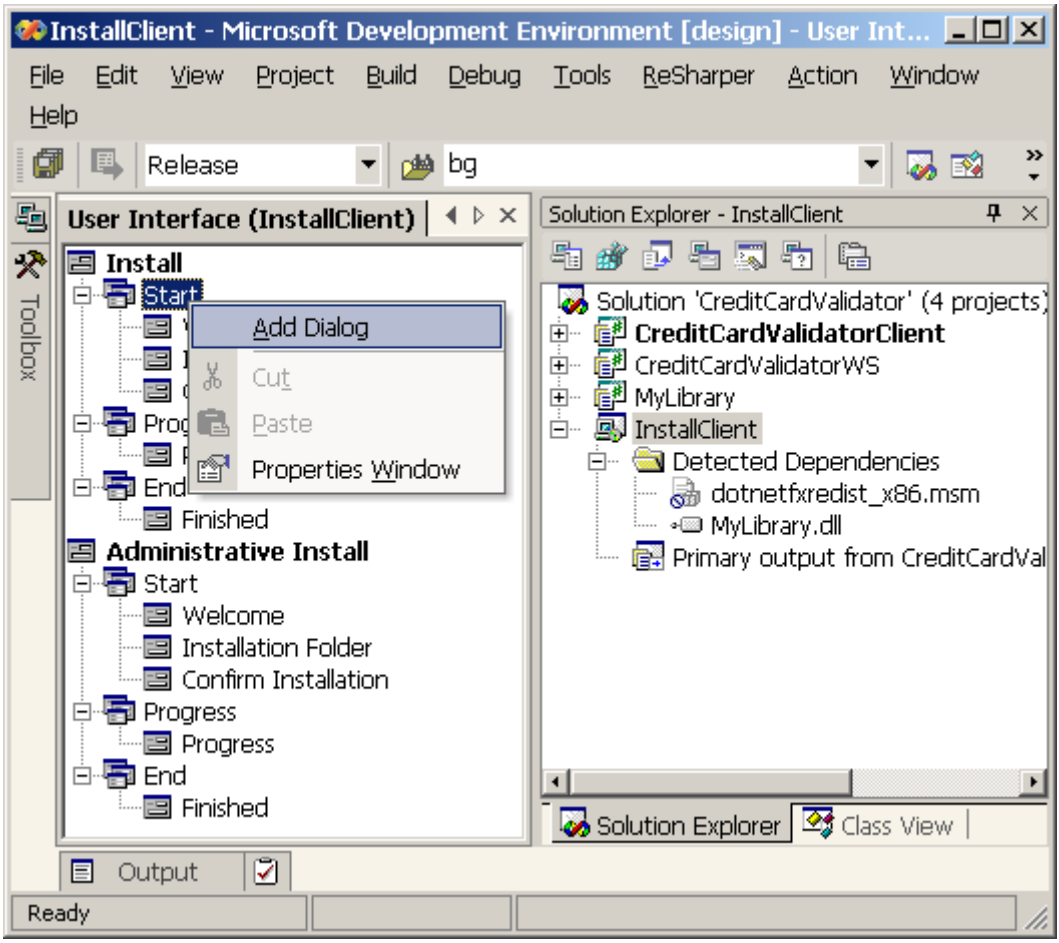
Добавяне на препратка на работния плот (Desktop)

Добавянето на препратка на работния плот технически не се различава много от Добавяне на препратки в Start Menu на Windows.

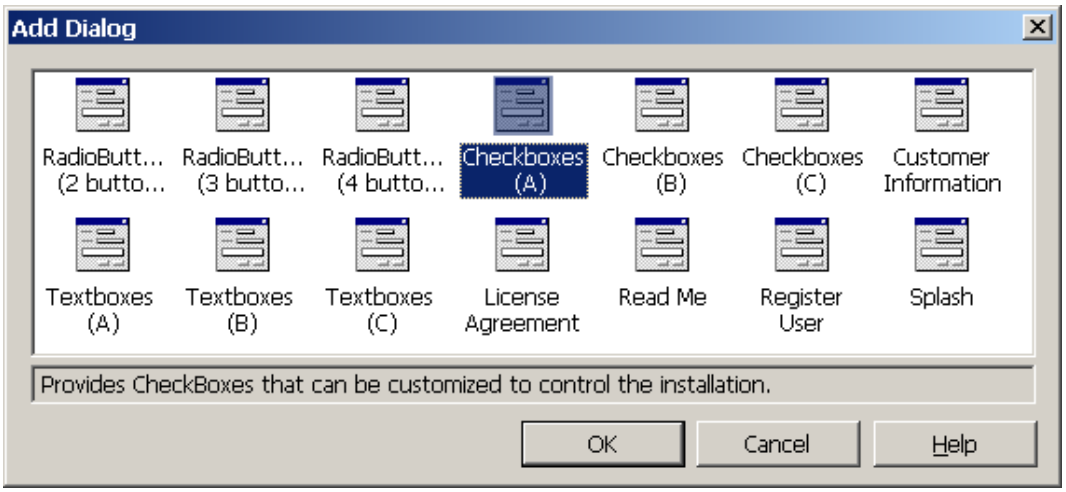
В указанията на Майкрософт за Windows базираните приложения ("Designed for Microsoft Windows XP" Application Specification – http://www.microsoft.com/ winlogo/software/downloads.mspx) е посочено, че поради претрупване на работния плот следва поставянето на икони да се извършва само по изрично указание на потребителя. Технически това означава добавяне на диалогов прозорец, в който потребителя да укаже предпочитанията си. Като следваме указанията от точката Добавяне на препратки в Start Menu на Windows, добавяме икона в папката **`User's Desktop`**.

Показваме изгледа **`User Interface`** – от контекстното меню на инсталационния проект **`InstallClient`**, избираме View -> User Interface. В него е показана последователността на диалозите, които се показват по време на нормална инсталация и административна инсталация.

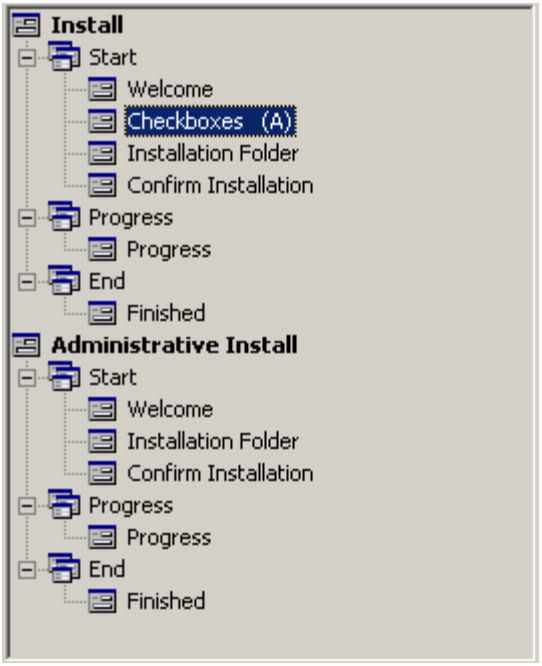
От контекстното меню на диалога Start избираме Add Dialog:



Показва се диалоговия прозорец Add Dialog, от който можем да изберем шаблон, съдържащ контролите, от които имаме нужда:



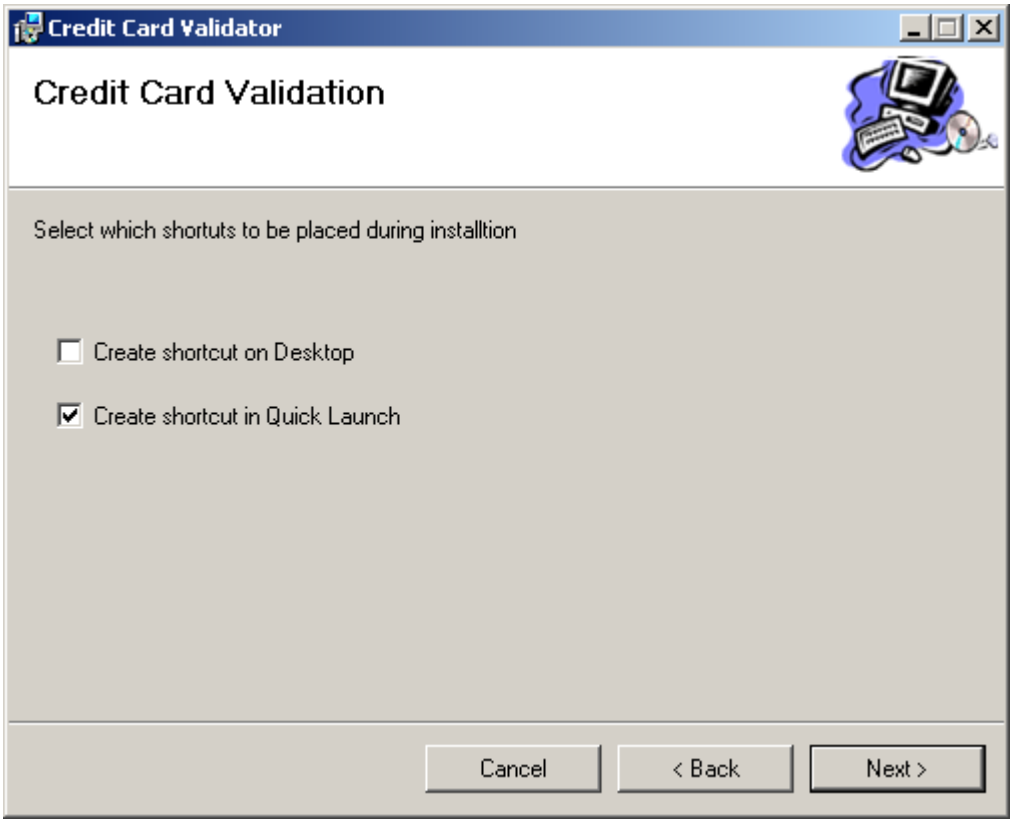
Избираме шаблона **`CheckBoxes (A)`** и потвърждаваме с бутона **`[OK]`**. Новосъздаденият диалог се добавя като последен в поредицата, но е необходимо да бъде преместен преди диалога **`Installation Folder`**. С левия бутон на мишката може да го влачим и пуснем върху диалога **`Welcome`** или от контекстното меню на **`CheckBoxes (A)`** да избираме позицията му чрез командите **`Move Up`** и **`Move Down`**. След това действие последователността на диалоговите прозорци трябва да е подобна на тази:



След като вече имаме подходящия диалог в поредицата, трябва да го настроим, за да показва подходящи съобщения. От прозореца с неговите свойства въвеждаме както следва:

Свойство	Стойност
`Banner Text`	Credit Card Validation
`Body Text`	Select which shortuts to be placed during installtion
`CheckBox1Label`	Create shortcut on Desktop
`CheckBox1Property`	`CHECKBOX_DESKTOP`
`CheckBox2Label`	Create shortcut in Quick Launch
`CheckBox2Property`	`CHECKBOX_QUICKLAUNCH`
`CheckBox2Value`	`Checked`
`CheckBox3Visible`	`false`
`CheckBox4Visible`	`false`

Ето как изглежда този диалогов прозорец в действие:



Вече имаме функциониращ диалогов прозорец в поредицата на нашия MSI пакет. Имаме и зададени променливи, които пазят избора на потребителя (`[CHECKBOX_DESKTOP]` и `[CHECKBOX_QUICKLAUNCH]`). Сега трябва да ги добавим като условие за инсталиране.

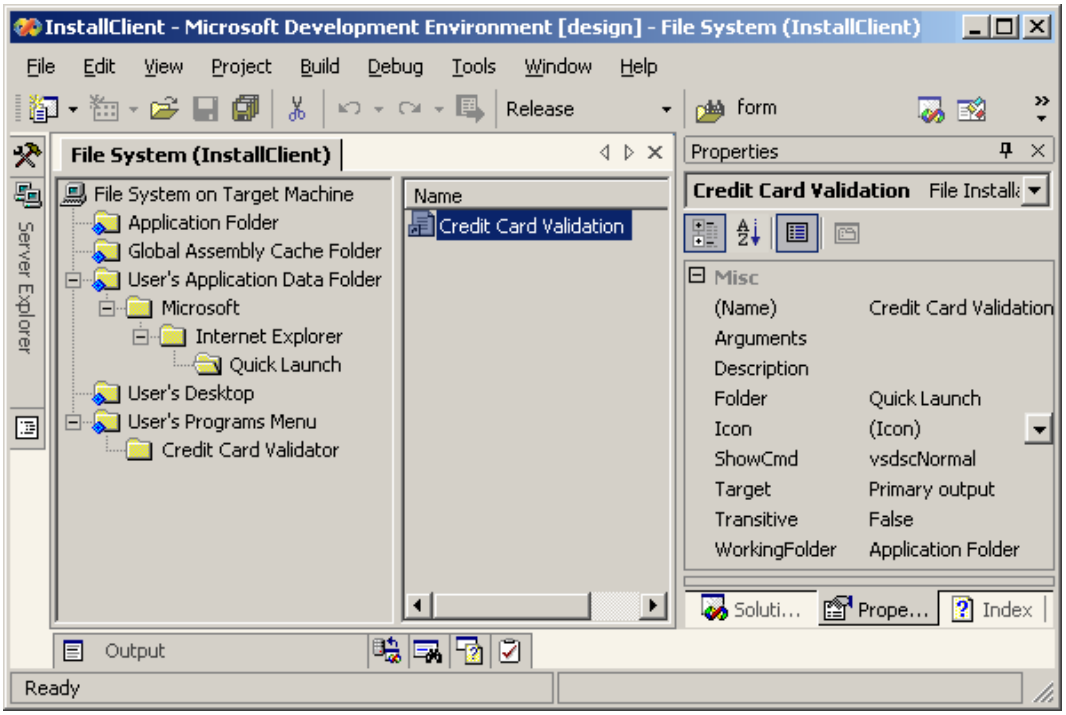
От изгледа **File System** избираме **User's Desktop** отваряме прозореца с характеристиките (командата Properties от контекстното меню). За свойството **Condition** задаваме стойност "**[CHECKBOX_DESKTOP]=1**".

Добавяне на препратки в Quick Launch

Quick Launch е възможност на Internet Explorer да показва лента с препратки на работния плот на Windows. Пътят до тази директория по подразбиране е **C:\Documents and Settings\[UserName]\Application Data\Microsoft\Internet Explorer\Quick Launch** като на мястото на **[UserName]** стои името на текущия потребител. За щастие в променливите на Windows Installer съществува **User's Application Data Folder**, която извлича пътя до **C:\Documents and Settings\[UserName]\Application Data** на текущия потребител. Можем да добавим директорията **User's Application Data Folder** от контекстното меню File System on Target Machine -> Add Special Folder -> User's Application Data Folder.

След това от изгледа **File System** избираме **User's Application Data Folder** с десен бутон на мишката и от контекстното меню избираме Add -> Folder. За име въвеждаме **Microsoft**. От контекстното меню на новосъздадената папка повтаряме горното действие и създаваме поддиректория **Internet Explorer**. В нея създаваме поддиректория **Quick Launch**.

От контекстното меню на детайлната област избираме Create New Shortcut и повтаряме действията от точка "Добавяне на препратки в Start Menu на Windows". Като резултат екранът трябва да изглежда подобно на този:



За да може потребителят да избира дали да се създаде препратка в Quick Launch (чрез диалоговия прозорец, който създадохме в точка "Добавяне на препратка на работната площ (Desktop)") трябва да зададем инсталационно условие: избираме папката **User's Application Data Folder** и от прозореца с характеристиките въвеждаме за свойството **Condition** стойност "**[CHECKBOX_QUICKLAUNCH]=1**".

Препратка за деинсталиране на приложението

Като цяло създаването на препратка за деинсталиране на приложението не се препоръчва от Майкрософт и е залегнало в указания за приложения за Windows ("Designed for Microsoft Windows XP" Application Specification – http://www.microsoft.com/winlogo/software/downloads.mspx). Поради спецификата на добавяне на подобна функционалност и честите практики на софтуерните компании да поставят подобна препратка в стартовото меню ще го опишем. Това следва да се разглежда като пример за създаване на икони към приложения, които не се инсталират от текущия MSI пакет.

За да се запази целостта на MSI пакета, Windows Installer не позволява поставяне на препратки към файлове, които не се разпространяват с дадения MSI пакет. По този начин се предотвратява зависимостта на MSI пакетите към външни файлове. Както ще покажем в точката "Инсталиране/деинсталиране на MSI пакетите", основният команден файл на Windows Installer е **msiexec.exe**. Това е командата, към която Windows насочва обработката на MSI пакетите независимо от мястото, от което са инициирани (Control Panel, Windows Explorer или чрез команда).

Създаваме файл **Uninstall.bat** в директорията на проекта **CreditCardValidatorClient** със следното съдържание:

Uninstall.bat

```
msiexec /x {A3E3AA3C-3D63-4A5B-8F0A-B32BDEDCD4D8}
```

Последния параметър е от свойството **ProductCode** на инсталационния проект (вж. Характеристики на инсталационния проект).

От контекстното меню на проекта **InstallClient** избираме Add -> File… и избираме **Uninstall.bat**. След като вече имаме подходящ файл може да създадем препратка в стартовото меню към **Uninstall.bat**.

Създаване на инсталационен пакет на уеб услуга

След като създадохме инсталационен пакет за Windows приложението, остава да добавим и инсталационен пакет за уеб услугата.

От заредоното решение **CreditCardValidator.sln** във Visual Studio .NET 2003, избираме Add -> New Project и избираме Web Setup Project от категорията Setup and Deployment Projects. За име въвеждаме

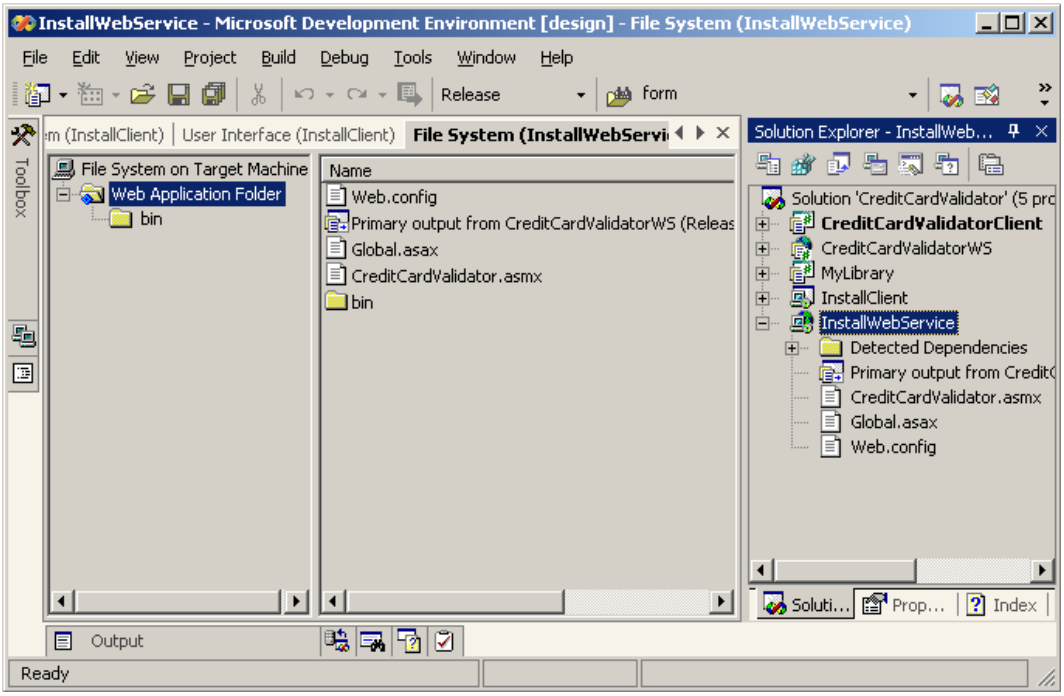
InstallWebService и избираме опцията **Add to Solution**. Потвърждаваме с бутона **[OK]**.

Новият проект се създава и се добавя към решението **CreditCardValidator.sln**.

Задаваме стойности на свойствата **Title** и **ProductName** "Credit Card Web Service" от прозореца с характеристиките на инсталационния проект, както направихме в точка Характеристики на инсталационния проект.

Както и в подточка Добавяне на файлове към инсталационния проект към Създаване на инсталационен пакет на Windows базирано приложение ще добавим проектните файлове към създадения инсталационен проект. От контекстното меню на проекта **InstallClient** избираме Add -> Project Output…

Както вече споменахме, Project Output включва създаваните по време на компилация файлове – **.exe**, **.dll** и **.config**. За да е функционална уеб услугата, трябва да добавим още фалове – **Web.config**, **Global.asax** и **CreditCardValidator.asmx**. За целта отваряме изгледа **File System** на инсталационния проект **InstallWebService** (от контекстното меню на **InstallWebService** избираме View -> File System). От контекстното меню на **Web Application Folder** избираме Add -> File… Отваря се стандартен **File Open Dialog**. Отваряме проектната директория на проекта **CreditCardValidatorWS** и избираме файловете **Web.config**, **Global.asax** и **CreditCardValidator.asmx**:



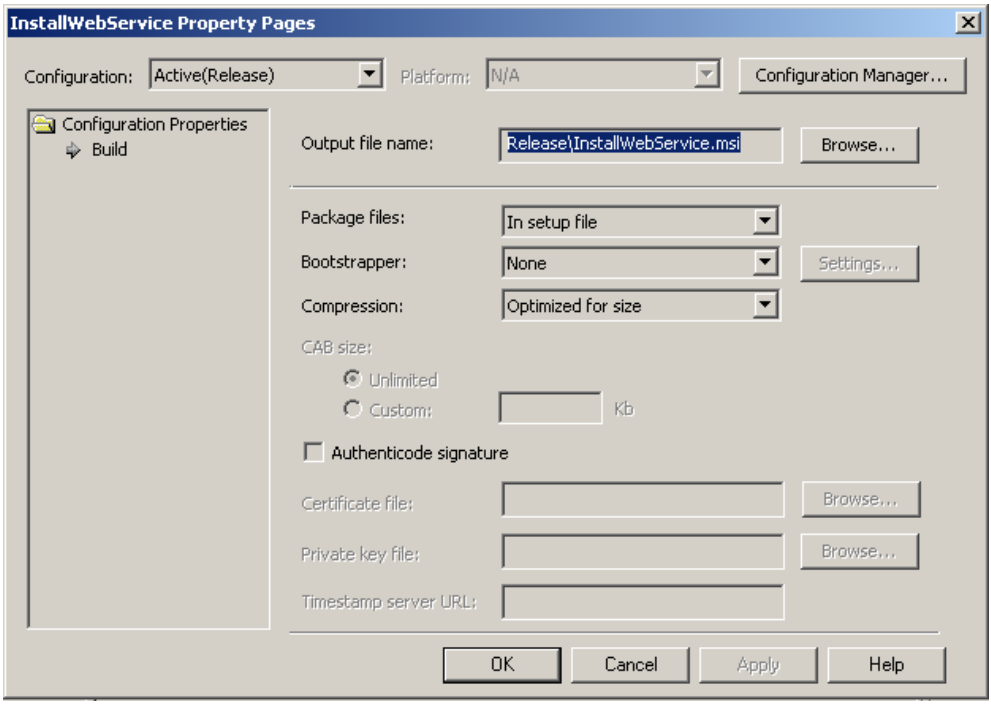
От прозореца със свойствата на **Web Application Folder** задаваме на **VirtualDirectory** стойност "**CreditCardValidatorWS**". Това е всичко.

Допълнителни настройки на инсталационните проекти във VS.NET 2003

За всеки от двата инсталационни проекта се създават по три файла след компилация на решението: **Setup.Exe**, **Setup.Ini** и **[име на проекта].msi** (**InstallWebService.msi** и **InstallClient.msi**).

Освен това по подразбиране файловете се пакетират с ниско ниво на компресия. За да променим това, от характеристиките на инсталационния проект (от менюто на Visual Studio .NET 2003 Project -> Properties) задаваме следните характеристики:

- **Bootstraper** – None
- **Compression** – Optimize for size



В резултат след компилация ще се генерират по един **.msi** файл за всеки от проектите, който ще бъде оптимално компресиран.

Инсталиране/деинсталиране на MSI пакетите

След като създадохме двата инсталационни пакета, е време да ги тестваме. Компилираме ги и намираме **.msi** файловете. Препоръчително е да използваме Virtual PC за тестовете, за да няма конфликти с вече инсталираните приложения и за да проверим дали всичко работи върху чиста

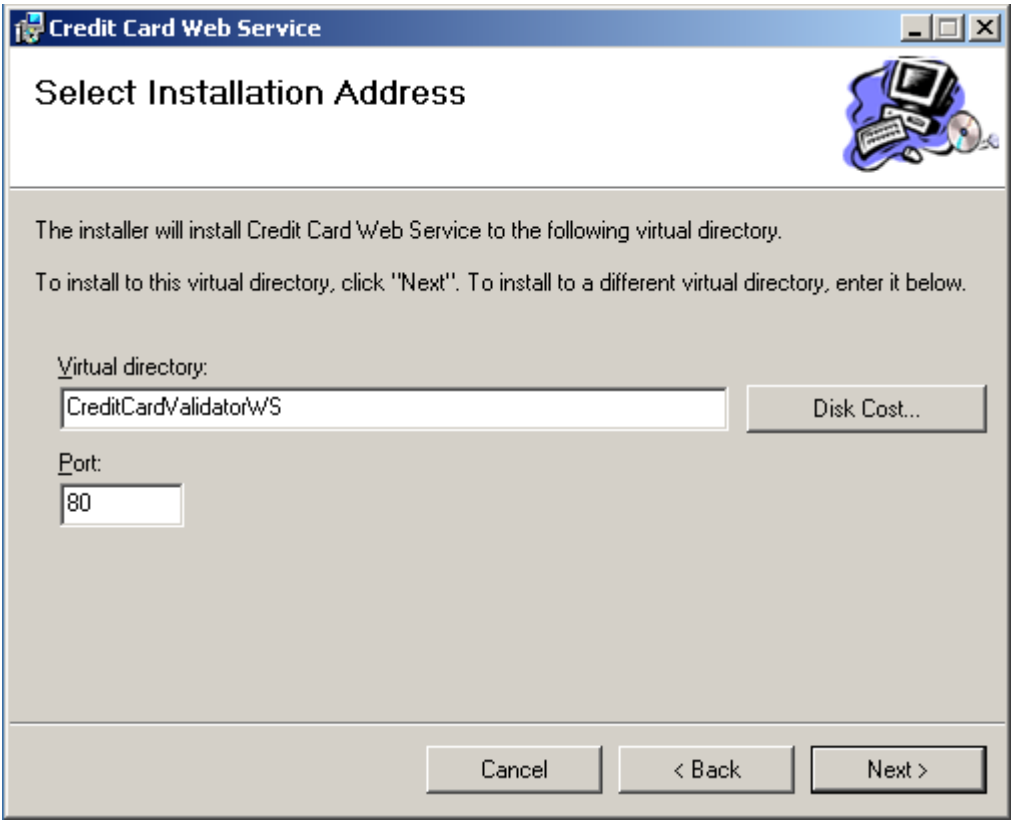
инсталация на Windows и IIS. Ако тестваме локално, ще се получи конфликт с **CreditCardValidatorWS**, тъй като **InstallWebService.msi** създава виртуална директория в IIS с име **CreditCardValidatorWS**.

Инсталацията на MSI пакет може да се стартира чрез двойно щракване на мишката върху **.msi** файла или чрез командата:

```
msiexec.exe /I InstallClient.msi
```

Първо инсталираме **InstallClient.msi**. Инсталацията протича по начин, който е добре познат на потребителите. Обърнете внимание, че препратките от **InstallClient.msi** са създадени в зависимост от избраните настройки в диалоговия прозорец, който добавихме в точка Добавяне на препратка на работната площ (Desktop).

Стартираме и инсталацията на уеб услугата от **InstallWebService.msi**. От втория диалогов прозорец имаме възможност да зададем настройките на виртуалната директория, която ще бъде създадена в IIS.



Не правим промени по тези настройки и завършваме инсталационния процес.

Ако сте направили промени в настройките на виртуалната директория в IIS от прозореца по-горе (или уеб услугата е инсталирана на отделен

компютър) трябва да се промени конфигурационният файл на клиентското приложение, за да може да се достъпи уеб услугата.

Ето го съдържанието на конфигурационния файл на **CreditCardValidatorClient**:

CreditCardValidatorClient.exe.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="webServiceURL"
value="http://localhost/creditcardvalidatorws/CreditCardValidator
.asmx"/>
  </appSettings>
</configuration>
```

На по-късен етап приложенията могат да се деинсталират чрез Control Panel -> Add/Remove Programs или чрез командата:

```
msiexec.exe /x InstallClient.msi
```

Упражнения

1. Създайте многомодулно асембли. Именувайте го силно. Разгледайте манифеста му.
2. Създайте проект **TestLibrary** от тип Class Library. Дайте силно име на резултатното асембли **TestLibrary.dll**. Добавете в проекта примерен клас и статичен метод, който връща в резултат текста "MyAssembly v.1.0.0.1". Използвайте атрибутите в **AssemblyInfo.cs**, за да зададете за асемблито версия **1.0.0.1**.
3. Създайте Windows базирано приложение **TestWinApp** и реферирайте от него асемблито от предходното упражнение (**TestLibrary.dll**) като частно асембли. Можете да направите това като от VS.NET добавите референция към проекта **TestLibrary**. При стартиране на проекта **TestWinApp** покажете диалогова кутия със съдържание върнатия от статичния от метод **TestLibrary** текст. След компилация би трябвало в изходната директория да имате файловете **TestWinApp.exe** и **TestLibrary.dll**.
4. Създайте поддиректория **assemblies** в директорията с компилираното приложение от предходното упражнение (**TestWinApp.exe**) и преместете в нея асемблито **TestLibrary.dll**. Добавете конфигурационен файл и задайте частни пътища за търсене на частните асемблита с тага **<probing>**. Посочете директорията **assemblies**. Приложението работи правилно, нали? Премахнете тага **<probing>** от конфигурационния файл и използвайте вместо него тага **<codebase>**. Тествайте отново.

5. Добавете асемблито от **`TestLibrary.dll`** в GAC. Изтрийте поддиректория **`assemblies`**. Приложението **`TestWinApp.exe`** работи нормално, нали? Деинсталирайте **`TestLibrary.dll`** от GAC. Приложението спря да работи, нали?
6. Променете текста, връщан от статичния метод от проекта **`TestLibrary`**, на "MyAssembly v1.0.0.2" и променете версията на **`1.0.0.2`**. Добавете новата версия на асемблито **`TestLibrary.dll`** в GAC. Приложението **`TestWinApp.exe`** все още не работи, нали?
7. Създайте Publisher Policy File, за да пренасочите **`TestLibrary.dll 1.0.0.1`** към **`TestLibrary.dll 1.0.0.2`**. Приложението **`TestWinApp.exe`** трябва отново да работи нормално.
8. Създайте проста система за събиране на числа, реализирана като уеб услуга с Windows базиран и уеб базиран клиент. Създайте инсталационни пакети за уеб услугата и за клиентските приложения. Инсталирайте ги върху друга машина и ги тествайте. Работят ли правилно? Деинсталирайте ги чрез Control Panel -> Add/Remove Programs. Работи ли правилно деинсталацията?

Използвана литература

1. Михаил Стойнов, Асемблита и Deployment – http://www.nakov.com/dotnet/lectures/Lecture-23-Assemblies-v1.0.ppt
2. Paul Slater, Deploying .NET Applications Lifecycle Guide, Microsoft Press, 2003, ISBN 0735618461
3. "Designed for Microsoft Windows XP" Application Specification – http://go.microsoft.com/fwlink/?LinkId=9775
4. Microvision Corporation's Website – http://www.InstallShield.com/
5. Wise Solutions's Website – http://www.wise.com
6. MSDN Library – http://msdn.microsoft.com
7. GotDotNet Website - http://samples.gotdotnet.com/quickstart/aspplus/
   - Working with Resource Files
8. Junfeng Zhang's Blog - http://blogs.msdn.com/junfeng
   - MultiModule Assemblies

Глава 26. Сигурност в .NET Framework

Автори

Тодор Колев

Васил Бакалов

Необходими знания

- Базови познания за .NET Framework
- Базови познания за езика C#
- Базови познания за работата на CLR, асемблита и атрибути

Съдържание

- **Сигурността в .NET Framework**
  - Безопасност на типовете и защита на паметта
  - Хващане на аритметични грешки
  - Application Domains
  - Симетрично и асиметрично кодиране. Цифров подпис
  - Силно-именувани асемблита
  - Технологията Isolated Storage
- **Code Access Security**
  - Политиките за сигурност в .NET Framework.
  - .NET Security Policy Editor
  - Права (Permissions)
  - Декларативно и програмно искане на права
  - "Stack Walk" и контрол над правата
- **Role-Based Security**
  - Автентикация и оторизация
  - Identity и Principal обекти. WindowsIdentity и WindowsPrincipal
  - Оторизация по Principal – декларативна и програмна
- **Криптография в .NET Framework**
  - Изчисляване на хеш стойност
  - Използване на симетрични криптиращи алгоритми
  - Използване на асиметрични криптиращи алгоритми

- Използване на цифрови подписи
- Подписване на XML (XMLDSIG)

В тази тема ...

В настоящата тема ще разгледаме аспектите, в които .NET Framework подпомага сигурността на създаваните приложения. Това включва както безопасност на типовете и защита на паметта, така и средствата за защита от изпълнение на нежелан код, автентикация и оторизация, електронен подпис и криптография. Ще бъдат разгледани технологиите на .NET Framework за защита на кода (Code Access Security, Role-Based Security, силно-именувани асемблита), както и библиотеките за работа с криптография (симетрични и несиметрични криптиращи алгоритми, хеширащи алгоритми) и цифрови подписи. Накрая ще бъде разгледан стандартът за цифрово подписване на XML документи (XML-Signature) и поддръжката му в .NET Framework.

Сигурността в .NET Framework

Създаването на сигурна и надеждна платформа е било основната цел при проектирането на .NET Framework. Това, което я отличава като такава, е фактът, че програмният код не разполага директно с ресурсите на машината, а бива управляван от Common Language Runtime (CLR). Това е причината .NET изпълнимият код да се нарича управляван код (managed code).

По време на изпълнение управляваният код непрекъснато се контролира от CLR и по този начин се осигурява максимална защита от възникване на грешки причинени от неправилно управление на паметта, неправилна работа с типове и указатели и други често срещани проблеми.

За осигуряване на сигурността на .NET кода CLR съдържа специализирани компоненти, които предоставят проверка на типовете (Type checker), управление на изключенията (Exception manager) и управление на сигурността на кода (Code Access и Role-Based Security).

Безопасност на типовете

Управляваният код е защитен от неправилна работа с типовете. На първо място това означава че кодът на .NET Framework приложенията не използва указатели към паметта. Вместо тях се използват така наречените референции към обекти. Те представляват едно много по-високо ниво на абстракция в сравнение с указателите и по този начин позволяват по-голям контрол от страна на CLR. Референциите са силно типизирани. Това означава, че не можем да присвоим референция от даден тип към несъвместим с него обект. Ако се опитаме да го направим, CLR ще генерира изключение **`System.InvalidCastException`**. Ето един пример:

```
object bytes = new byte[5];
char[] chars = (char[]) bytes;
// System.InvalidCastException is thrown
```

Достъпът до чужди обекти и области от паметта е ограничен. По този начин се осигурява както целостта на данните, така и безпроблемната работа на цялото приложение.

Проблемът "Buffer overrun"

Проблемът "buffer overrun" (или както е известен още "buffer overflow") възниква при препълването на масиви и символни низове. Това води до осъществяване на достъп до оперативна памет извън отделената за даден масив или символен низ и съответно до непредвидимо поведение на програмния код.

Проблемът "buffer overrun" може да доведе до сериозно компрометиране на сигурността, защото в някои случаи позволява "инжектиране" и

изпълнение на чужд код в контекста на приложението. Такава атака се осъществява като посредством препълване на масив се презапише адресът за връщане от последния извикан метод в стека за изпълнение на приложението и се пренасочи към код, подаден от атакуващия. Ако атаката е успешна, нападателят може да получи правата, с които се изпълнява приложението.

В .NET Framework този проблем е решен чрез вградена в CLR защита на масивите и символните низове от препълване. При опит за достъп до елемент от масив или низ, който е след неговия край или преди неговото начало, възниква изключение (exception), което прекъсва изпълнението на програмата и е възможно да бъде прихванато и обработено.

Следващият програмен фрагмент демонстрира възникване на изключение **`System.IndexOutOfRangeException`**, породено от опит за достъп до елемент, който е извън обработвания масив:

```
private static void CopyArray(byte[] aSrc, byte[] aDest,
   int aSize)
{
   for (int i=0; i< aSize; i++)
      aDest[i] = aSrc[i];
}

static void Main()
{
   byte[] arr1 = new byte[10];
   byte[] arr2 = new byte[5];
   CopyArray(arr1, arr2, 10);
   // System.IndexOutOfRangeException is thrown
}
```

Защита на паметта

При създаване обекти в .NET Framework те се разполагат в динамичната памет, т. нар. managed heap, който се управлява от CLR. По този начин цялата отговорност по заделяне на нова памет и освобождаването ѝ се поема от CLR.

Неизползваните обекти се почистват автоматично от т. нар. Garbage Collector. През определен интервал от време или след изрично извикване той почиства паметта от всички обекти, към които няма референции и съответно не са необходими повече за работа на приложението.

Всичко това прави почти невъзможно възникването на някои от най-неприятните проблеми в програмирането, свързани със загубата на памет (memory leaks). Проблемът с използването на неинициализирана памет също е решен благодарение на CLR, който се грижи новосъздадените променливи в .NET Framework винаги са инициализирани и занулени.

Прихващане на аритметични грешки

При работа с аритметични операции са възможни препълвания на типовете. Това става при получаване на резултат, който не се събира в типа, който е използван или при преобразуване на тип с по-голяма размерност към тип с по-малка размерност. Например ако имаме две числа от тип **sbyte** (200 и 150) и ги умножим едно с друго, резултатът не може да се побере в **sbyte**, въпреки, че самите числа 200 и 150 се събират. В този случай се получава препълване на типа **sbyte**.

В .NET Framework има вграден механизъм за прихващане на аритметични препълвания за целочислените типове. Ако се случи препълване на тип в проверявана (checked) част от кода възниква изключение, което прекъсва изпълнението на програмата и е възможно да бъде прихванато и обработено.

Проверяван и непроверяван код

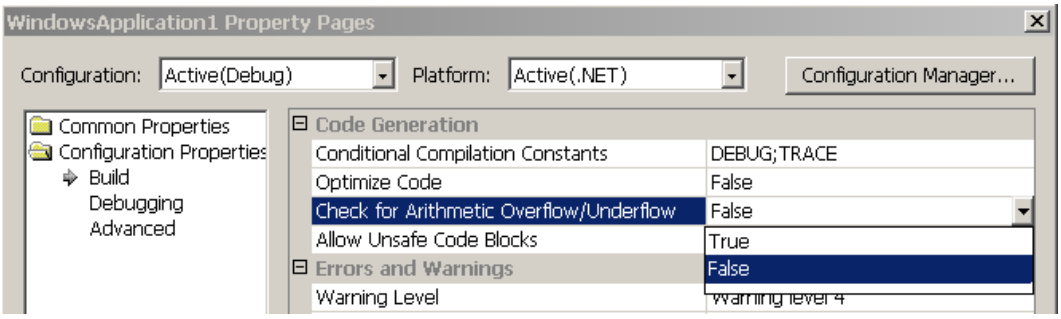
Възможно е програмистът да определя дали даден код да бъде проверяван (checked) или съответно непроверяван (unchecked). По подразбиране компилаторът на C# не проверява кода. Има два начина това да бъде променено. Единият е чрез промяна на настройките на компилатора, а другият е чрез изрично определяне на фрагменти от кода, които да се проверяват или съответно да не се проверяват.

Опции на компилатора за проверка на аритметиката

Използването на C# компилатора от командния ред в режим на проверяване за аритметични препълвания се извършва чрез опцията **/checked+**, а режим на непроверяване се указва с **/checked-**:

```
csc /checked+ SomeFile.cs
csc /checked- SomeFile.cs
```

Тази настройка може да се задава и от Visual Studio .NET 2003 от формата за настройки на съответния проект, като се промени стойността на полето **Check for Arithmetic Overflow/Underflow** на **True** или **False** съответно за проверяване или непроверяване на кода.



Ключовите думи checked и unchecked

Другият вариант за определяне дали кодът да се проверява или не за аритметични препълвания е чрез ключовите думи **checked** и **unchecked** в C#. Те задават област (блок) от код в който да се извършва или съответно да не се извършва проверка. Указаното с тези ключови думи не се влияе от настройките на компилатора. Следващият програмен фрагмент демонстрира определянето на код който да бъде проверяван за аритметични препълвания:

```
checked
{
   int a = 250000;
   int square = a*a;
   // System.OverflowException is thrown
}
```

Изпълнението на този код предизвиква възникването на изключение **System.OverflowException** поради препълване на типа **int**.

Application Domains

Операционните системи обикновено предлагат механизми за изолиране на приложенията едно от друго. Това изолиране е нужно, за да се предпазят кодът и данните на дадено приложение от това да бъдат неправомерно повлиявани от работата на друго приложение.

Изолирането на приложенията в операционната система Windows се реализира посредством стартирането на всяко отделно приложение в отделен процес и защита на паметта на ниво процесор. Това осигурява нужното изолиране, но затруднява комуникацията между приложения. Адресите от паметта са относителни за всеки процес и поради тази причина указател от едно приложение (процес) не може да се използва в друго. Това налага обмяната на данни между приложенията (inter-process communication) да се извършва чрез посредник и специализиран протокол за комуникация, което понижава производителността.

Application domains предоставят по-ефективен и същевременно надежден начин за изолиране на .NET приложенията едно от друго. Всяко .NET приложение работи в един application domain, а няколко application domains могат да работят в един и същ процес на операционната система. По този начин се постига желаната изолация по памет, данни и код и в същото време се предоставят много по-гъвкави и бързи средства за комуникация между .NET приложенията защото не се налага сложна комуникация между процеси. Когато няколко .NET приложения се стартират в един процес, се спестяват много ресурси, защото CLR се зарежда и инициализира само веднъж, а не толкова пъти, колкото са приложенията.

Фактът, че Application domains се управляват от CLR дава възможност да се настройват правата, с които разполага всеки един Application domain.

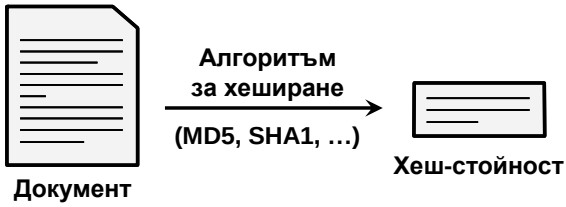
Така се постига повишена сигурност при изпълнение на дадено асембли и се дава възможност едно приложение да зареди и изпълни дадено .NET приложение с ограничени права.

Основни криптографски понятия

Преди да преминем към подписването на асемблита, ще изясним някои общи криптографски понятия, които ще използваме по-нататък в настоящата тема.

Хеширане

Хеширането се използва, за да се съпостави на данни с произволна дължина, число (хеш, хеш-стойност, хеш-код), което е тяхна уникална необратима репрезентация. Това означава, че имайки хеш-стойността не можем да възстановим първоначалните данни и хеширайки едни и същи данни всеки път ще получаваме един и същи хеш.



Уникалността на хеш-стойността означава, че за различни данни ще получаваме различни хеш стойности. Тази уникалност обаче не е абсолютна, тъй като в общия случай входните данни имат по-голяма дължина от хеш стойността и следователно поне два различни набора от входни данни ще имат един и същи хеш (според принципа на Дирихле за чекмеджетата). Когато това се случи, казваме, че има колизия.

Някой от най-известните алгоритми за хеширане са MD5, SHA1, SHA256, SHA384 и SHA512. При тях е доказано, че колизии съществуват, но е много трудно да се постигнат на практика. Тези алгоритми се наричат криптографски силни хеширащи алгоритми. Те имат свойството, че по дадена хеш стойност е изключително трудно да се намерят данни, от които тя се получава.

Симетрично криптиране

Симетричното криптиране се нарича още криптиране със секретен ключ. При него криптирането и декриптирането се извършват с един и същи ключ, който не трябва да бъде известен на никой освен на страните, включени в обмена на информация. Това налага те да се споразумеят за използвания ключ преди да започне обмена на информация, без възможност друга страна да го разбере. На схемата е показан типичният сценарий за криптиране и декриптиране със симетричен ключ:

Симетричното криптиране е подходящо да се използва в среди с един потребител, например за да защитим с таен ключ достъпа до определена директория на диска. По този начин липсва опасността ключът да бъде прихванат по време на споразумението между две комуникиращи страни. Този тип криптиране работи с висока скорост и има възможност за работа с потоци от данни.

Някой от най-известните алгоритми за симетрично криптиране са DES, 3DES, RC2, RC4, RC5, Blowfish и IDEA, като в .NET Framework са имплементирани DES, 3DES, RC2 и Rijndael/AES (Advanced Encryption Standard).

Асиметрично криптиране

Асиметричното криптиране, наричано още криптиране с обществен (публичен) ключ, работи не с един, а с двойка съответни **публичен** и **личен** (частен) ключ (public/private ключове). Публичният ключ се използва за криптиране на съобщението, а декриптирането е възможно единствено със съответния личен ключ. Той не може да бъде извлечен от публичния ключ.

Асиметричното криптиране има едно основно предимство пред симетричното криптиране - при него не се налага предаване на ключа преди започване предаването на информация и съответно отпада възможността той да бъде разбран от трета страна. Това предимство не идва безнаказано, тъй като за да функционира правилно схемата трябва да има изградена система-хранилище, в която да се пазят двойките ключове и публичните ключове да се предоставят на всички страни. Само така можем да имаме сигурност, че ключът е наистина на лицето, което твърди, че е негов притежател, и удобен начин да намерим публичният ключ на някой, на когото искаме да изпратим тайна информация. Без гаранция за принадлежността на един публичен ключ, той губи своето значение.

Асиметричното криптиране е по-сигурно от симетричното, от гледна точка че не се налага да се предава парола (ключ), но за сметка на това е много по-бавно и изисква повече изчислителни ресурси. При него съществува и ограничение на максималната дължина на криптираното съобщение в

зависимост от дължината на използвания ключ. Това го прави неподходящо за обработване на потоци от данни, но много подходящо за размяна на симетрични ключове или други споделени тайни.

Често пъти в практиката се ползва комбинация от несиметрични и симетрични алгоритми за криптиране. Например при SSL (Secure Socket Layer) протокола се използва криптография с публичен ключ, за да се обмени по сигурен начин т. нар. сесиен ключ, който се използва след това за симетрично криптиране и декриптиране на обменяните данни.

Разпространени алгоритми за асиметрично криптиране са RSA, DSA, Diffie-Hellman, ECDSA (Elliptic-Curves DSA).

Често пъти в практиката се ползва комбинация от несиметрични и симетрични алгоритми за кодиране. Например при SSL (Secure Socket Layer) протокола се използва криптография с публичен ключ, за да се обмени по сигурен начин т. нар. сесиен ключ, който се използва след това за симетрично кодиране и декодиране на обменяните данните.

Public-Key Infrastructure (PKI)

Public-key infrastructure (PKI) е комбинацията от софтуер, технологии за криптиране и услуги, която позволява осъществяването на сигурната комуникация базирана на цифрови подписи и публични ключове. Тя позволява да се изгради доверие между непознати комуникиращи си страни посредством т. нар. "цифрови сертификати".

Цифровите сертификати са електронни документи, гарантиращи самоличността на дадено лице и удостоверяващи, че то е собственик на даден публичен ключ. Те се издават при строги мерки за сигурност от специални организации, на които се има доверие (сертификационни организации) и така се гарантира тяхната достоверност. В практиката за целите на електронния подпис най-масово се използват X.509 сертификати.

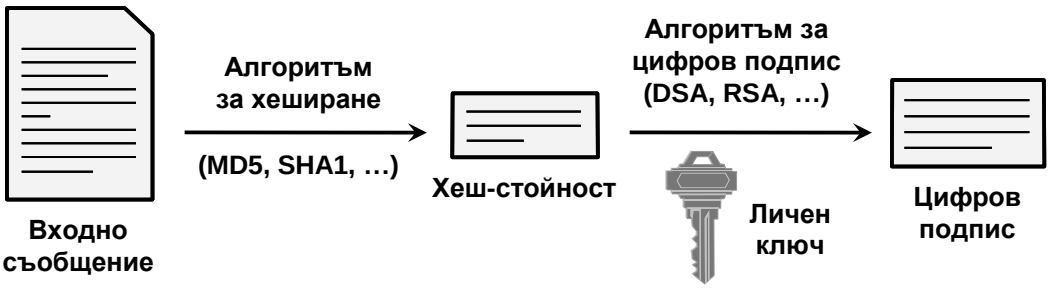
Цифров подпис

Цифровото подписване представлява механизъм за удостоверяване на произхода и целостта на информацията, предавана по електронен път. То е едно от приложенията на асиметричното криптиране, но за разлика от него, при цифровото подписване първоначалното криптиране се извършва с личния ключ, а декриптирането с публичния. Личният ключ се знае само от подписващия и не може да бъде извлечен от съответния му публичен ключ. Това гарантира самоличността на изпращача на подписаният документ или съобщение.

Полагане на цифров подпис

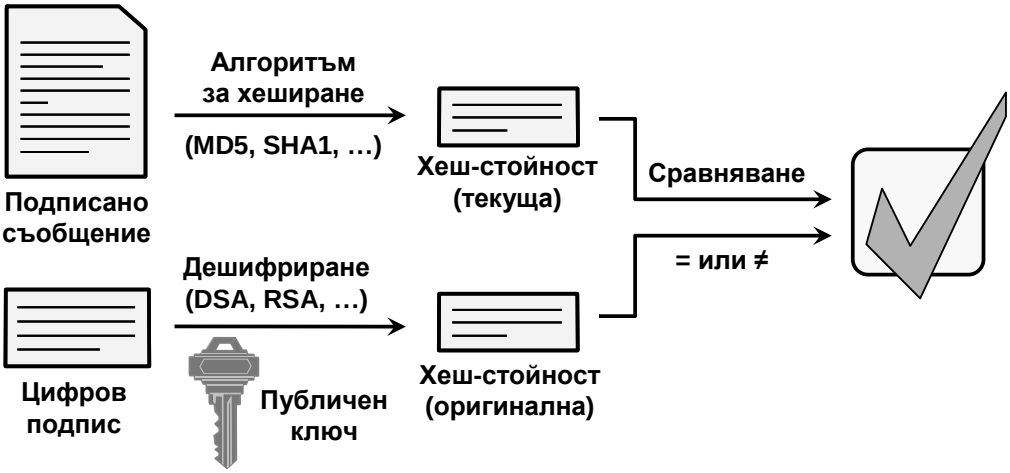
При процеса на цифрово подписване на даден документ към него се добавя допълнителна информация, наречена цифров подпис. Той трябва да осигури, че документът е подписан от точно определено лице. Затова цифровият подпис представлява хеш стойността (уникална репрезента-

ция) на подписвания документ, криптирана с личния ключ на подписващия.



Верификация на цифров подпис

Проверката се извършва чрез публичния ключ, който съответства на всеки личен ключ. Този публичен ключ се предава заедно с подписания документ или се извлича от централно хранилище, като самоличността на притежателя му се гарантира посредством цифров сертификат или по друг начин. Проверяващият генерира хеш стойност на получения документ и посредством публичния ключ декриптира електронния подпис, за да получи оригиналната хеш стойност на подписания документ. Ако двете хеш стойности съвпаднат, то цифровият подпис за дадения документ е валиден.



Силно-именувани асемблита

Силното име на едно асембли го идентифицира по уникален начин и гарантира, че то не е променяно след компилацията си. Силното име включва самото име на асемблито, версия, култура, цифров подпис и съответния му публичен ключ. Символният формат на силното име изглежда така: **`<име на асемблито>`**, **`<версия - major.minor.build.revision>`**, **`<култура>`**, **`<публичен ключ>`**. Това име за **`System.dll`** във

версия 1.1 на .NET Framework например е **System, Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b77a5c561934e089**.

Силното-именуваните асемблита са подписани. Това осигурява, че те не са променяни след тяхната компилация. За проверката на подписа от CLR се ползва публичният ключ от силното име. Той обаче не гарантира самоличността на производителя на асемблито. Това може да стане чрез цифров сертификат, който се предоставя отделно.

Силното име указва по уникален начин версията на асемблито. Уникалността му се определя от всеки един от неговите компоненти. Дори две асемблита да имат еднакво име, версия и култура те пак няма да имат еднакви силни имена, тъй като двойката от публичен и личен ключ използвани за подписването им ще са различни.

Уникалността на версиите на силно-именуваните асемблита дава възможност няколко версии на асембли с едно и също базово име да се инсталират и използват независимо. Тази уникалност позволява също едно .NET приложение да работи точно с версията на компонентите си, която очаква, а не с по-стари или по-нови версии. За да се осигури това и за всички подкомпоненти на даден компонент, всяко силно именувано асембли може да има референции само към други силно именувани асемблита.

При добавяне на референция от някое асембли **A.dll** към силно-именувано асембли **S.dll**, публичният ключ на **S.dll** се записва в компилираното асембли **A.dll**. Така **A.dll** се свързва само с конкретната версия на асемблито **S.dll** и само тази версия е възможно да бъде заредена и изпълнена. Всяко асембли, което използва дадено силно-именувано асембли, се сдобива с неговия публичен ключ по време на компилацията си и така няма нужда да се използва цифров сертификат.

Създаване на силно именувано асембли

За да се създаде силно име на едно асембли е нужно то да има зададена версия и култура и след това да бъде цифрово подписано. За да се извърши подписването е нужна двойка публичен и личен ключ. Такава двойка може да се генерира с помощния инструмент **Sn.exe** от .NET Framework SDK. Ето как от командния ред с него се генерира **.snk** файл съдържащ нужната двойка ключове:

```
sn –k MyKeyPair.snk
```

Самото подписване се извършва от компилатора на C#. Нужно е единствено да укаже в стандартно генерирания от VS.NET файл **AsemblyInfo.cs** пътя до файла съдържащ двойката публичен и личен ключ:

```
[assembly: AssemblyKeyFile(@"..\..\MyKeyPair.snk")]
```

Global Assembly Cache

Всеки компютър, който има инсталиран CLR разполага с общодостъпно на ниво машина място за съхранение на асемблита, наречено Global Assembly Cache (GAC). Там се съхраняват асемблита, които се налага да се използват от повече от едно .NET приложение. Поради това, че всички асемблита в GAC са общодостъпни е нужно те да са силно именувани, за да може всяко приложение да работи само с желаната от него версия.

За да бъде едно асембли съхранявано в GAC то трябва да бъде изрично инсталирано (добавено) в GAC. Това става с помощта на инструмента **`GacUtil.exe`**, които е част от .NET Framework SDK. Ето как от командният ред се инсталира асембли в GAC:

```
gacutil –i MyAssembly.dll
```

Премахването (деинсталирането) на асембли от GAC става чрез:

```
gacutil –u MyAssembly
```

Технологията Isolated Storage

Isolated Storage хранилищата представляват място на твърдия диск, което е предоставено само на дадено приложение. То се определя и управлява от CRL, като обемът на достъпното дисково пространство е ограничен. Използването на Isolated Storage хранилища е възможно и от .NET контроли в уеб страници в Интернет и други приложения с ограничени права.

Най-честата употреба на Isolated Storage хранилищата е за съхранение на потребителски настройки и кеширане на данни. Въпреки че предоставя място на диска, което не е достъпно за никое друго приложение, Isolated Storage не осъществява криптиране на информацията и не е подходящ за съхранение на чувствителна информация.

Isolated Storage хранилищата за данни могат да имат обхват, който определя за кого те са достъпни. Обхватът може да се определя спрямо потребител, асембли и домейн, като е възможна комбинация между критериите. Под домейн се разбира мястото (URL или локална директория), от което е заредено асемблито. Ето как се реализира достъп до хранилище достъпно за текущия потребител и текущото асембли:

```
IsolatedStorageFile store =
  IsolatedStorageFile.GetStore(
    IsolatedStorageScope.User |
    IsolatedStorageScope.Assembly,
    null, null);
```

След като получим обекта на хранилището можем да работим с файловете и директориите в хранилището по начин, подобен на стандартната работа

с файловата система. Отварянето на файл за четене се извършва със следния код:

```
IsolatedStorageFileStream stream =
  new IsolatedStorageFileStream(
  "notes.txt", FileMode.Open,
  FileAccess.Read, store);
```

Реалното разположение на Isolated Storage хранилищата е в:

```
C:\Documents and Settings\<username>\Local Settings\Application
Data\IsolatedStorage\...
```

Сигурност на кода (Code Access Security)

Сигурността на кода (Code Access Security – CAS) е фундаментален елемент на .NET Framework. Тя надгражда системата за сигурност на операционната система, като чрез нея се дава възможност да се управляват и ограничават правата, с които разполага дадено .NET приложение. Правата дадени чрез инструментите на Code Access Security са винаги по-малки или равни на правата на текущия потребител на операционната система, които използва даденото .NET приложение. CLR не може да даде на едно приложение права по-големи от тези на текущия потребител.

Политиките за сигурност в .NET Framework

Сигурността на кода (Code Access Security) се управлява от политики за сигурност (Security Policy). Политиките за сигурност определят групи код (Code Groups) на базата на доказателства (Evidences) за техния произход и задават набора права, с които разполагат асемблитата, попадащи в съответната група. Доказателствата за произход могат да бъдат:

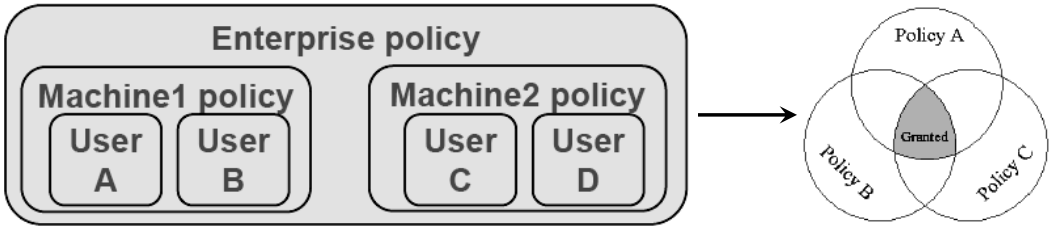
- Силно име на асемблито
- URL, от където идва асемблито
- Интернет зона, от където идва асемблито
- Хеш-стойност на асемблито

Групи права (Permission sets)

За улесняване процеса на задаване на права те се обединяват в предварително дефинирани набори от права, наречени именуваните списъци с права (Permission Sets). Стандартно в .NET Framework съществуват няколко системни списъци с права, които не могат да бъдат променяни и изтривани. Такива са: **`FullTrust`** – пълни права, **`Nothing`** – никакви права, **`Execution`** – права само за изпълнение и т.н. Освен тези предварително зададени списъци може да се създават и нови. Възможните права които могат да бъдат обединявани в тях ще бъдат разгледани по-долу.

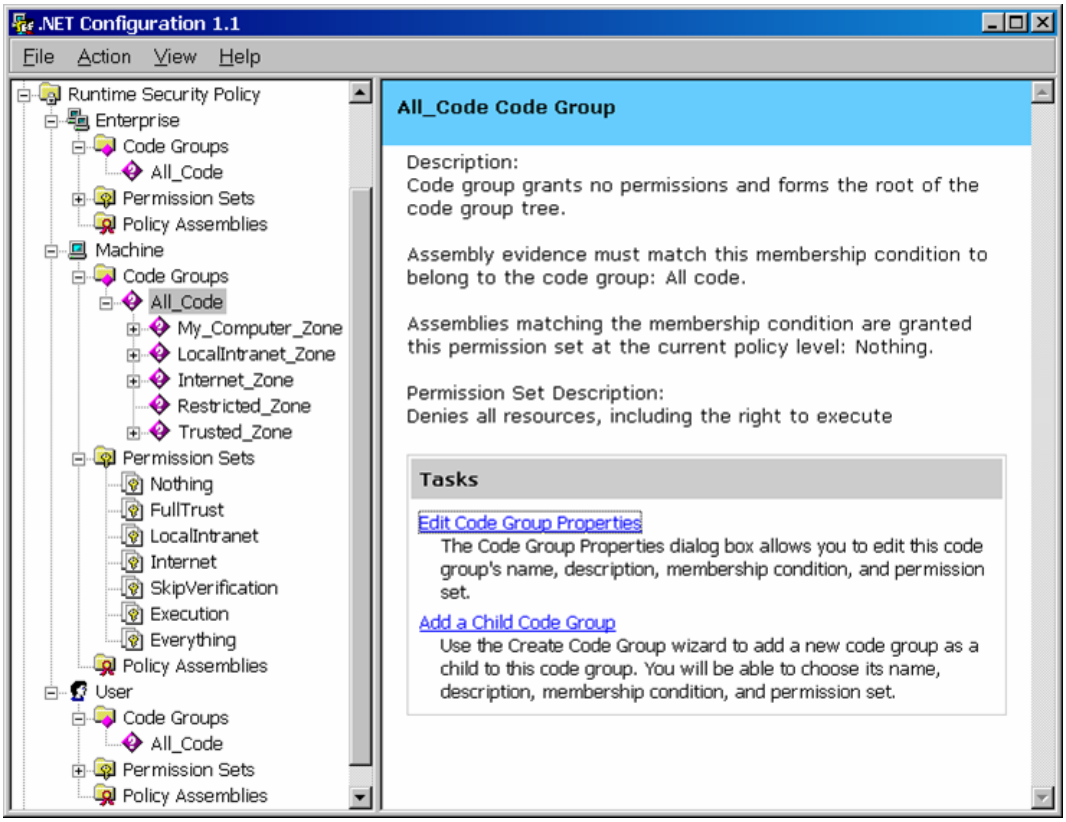
Нива на политиките за сигурност

Политиките за сигурност могат да бъдат определяни на три нива. Това са ниво Enterprise за целия Windows Domain, ниво Machine за текущата машина и ниво User за текущия потребител. Правата, с които разполага дадено асемблиде се определят като сечение на правата определени от трите нива на политиките. Това означава, че асемблито ще получи само правата, които му се предоставят и от трите нива едновременно:



Задаване на политика за сигурност

Политиките за сигурност и техните компоненти се администрират с помощта на инструмента за конфигуриране на .NET Framework:



За Windows XP/2000/2003, той се намира в **Control Panel | Administrative Tools | Microsoft .NET Framework Configuration 1.1.**

Права (Permissions)

Използвайки технологията Code Access Security, всяко едно асембли може да изисква или отказва права. Изискването на права от своя страна може да бъде изискване на **задължителни** права или изискване на **незадължителни** права.

Задължителни права

Ако някое от изисканите задължителни права не може да бъде предоставено на асемблито, то не бива заредено и възниква изключение **`System.Security.SecurityException`**. Това е по-приемливо от алтернативата асемблито да бъде заредено, но да не може да изпълнява функционалността си и потребителя да не е уведомен за това. При указването на задължителни изисквания потребителя бива уведомен за това, че даденото приложение не получава всички необходими му права. Така администратора на системата може своевременно да вземи мерки за осигуряване на нужните права.

Незадължителни права

Липсата на правата, изискани като незадължителни, не спира асемблито да бъде заредено и не предизвиква възникване на изключение. Незадължителните права, са права, които не се нужни за осъществяването на основните функции на приложението (асемблито). Асемблитата сами са отговорни да предвидят ситуациите, в които нямат обявените от тях като незадължителни права и да уведомят потребителя за това.

Отказани права

Отказаните права, са права, които асемблито изрично посочва, че не иска да му бъдат предоставени, независимо от активната политика за сигурност (security Policy). Това се използва, за да може асемблитата да си осигурят, че няма да имат повече от нужните им за нормална работа права. По този начин се избягва възможността даденото асембли да бъде използвано неправилно или недоброжелателно.

По-важни класове права в .NET Framework

Ето някой от основните класове реализиращи права, които са част от платформата Code Access Security:

Право	Описание
`FileIOPermission`	Четене / писане по файловата система
`IsolatedStorageFilePermission`	Достъп до изолирана виртуална файлова система тип "IsolatedStorage"
`UIPermission`	Използване на Windows Forms GUI
`FileDialogPermission`	Достъп до диалога за избор на файл

PrintingPermission	Печатане на принтер
WebPermission	Достъп до уеб ресурси
SocketPermission	Работа със сокети
OleDbPermission, **SqlClientPermission**	Достъп до база данни през OleDb или SqlClient доставчиците
RegistryPermission	Достъп до Windows Registry
ReflectionPermission	Достъп до Reflection

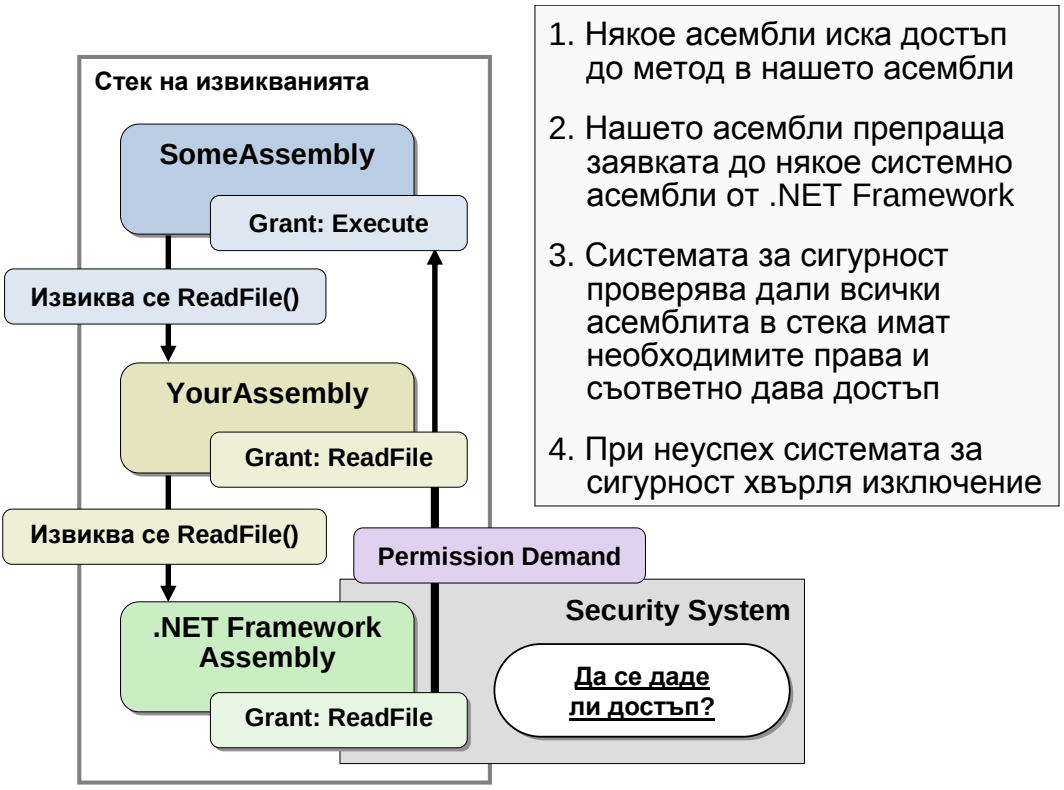
"Stack Walk" и контрол над правата

Правата в Code Access Security се определят поотделно за всяко асембли. Това дава възможност методите от асембли, имащо права за достъп до даден ресурс, да бъдат викани от методи на друго асембли, което няма тези права. По този начин е възможен неправомерен достъп до ресурси, като се използват правата на друго асембли.

Поради тази причина Code Access Security предоставя възможност всеки метод да проверява дали извикващите го методи имат нужните права. Тъй като права могат да се определят само до ниво асембли, всички методи, намиращи се в едно асембли, имат правата дадени на това асембли.

На схемата по-долу е показана работата на Stack Walk. За да се проверят правата на всички методи, които викат даден метод, е нужно да се обходи стекът на извикванията (call stack) - оттам идва и името Stack Walk. Обхождането и съответно проверката започват от метода, който непосредствено извиква метода, предизвикал тази проверка. Самият той не бива проверяван.

Обхождането на стека (Stack Walk) може да бъде контролирано от всеки един от методите, които биват обхождани. Това е възможно, само ако те самите притежават изискваното право. Методите могат да укажат, че даденото право, което се изисква, трябва да се притежава от всички останали методи нагоре в стека и съответно трябва да бъде потвърдено или някой от методите да укаже, че то не се притежава и проверката да завърши с отрицателен резултат. И при двата варианта проверката се прекъсва.



Контролът на обхождането на стека може да се извърши чрез извикване на следните методи на обект от клас, реализиращ права: **`Assert`**, **`Deny`**, **`PermitOnly`**.

- **`Assert`** - указва, че изискваното право трябва да бъде потвърдено.
- **`Deny`** - указва правото да бъде отказано.
- **`PermitOnly`** – указва, че всички останали права освен даденото трябва да бъдат отказани.

Декларативно и програмно искане на права

Управляваният код може да иска определени права за своето изпълнение по два начина: декларативно (статично) и програмно (динамично).

Декларативно искане на права

Декларативното искане на права се извършва чрез атрибути на ниво асембли. С тях се указва какви задължителни и незадължителни права изисква асемблито и кои права трябва да бъдат отнети (когато ги има).

Възможните действия на атрибутите за декларативно искане на права съответстват на стойности от изброения тип **`SecurityAction`**. Те могат да бъдат:

- **RequestMinimum** – указва, че асемблито не може да работи без съответното право.
- **RequestRefuse** – указва, че асемблито иска зададеното право да му бъде отнето.
- **Demand** – указва, че всички асемблита от стека на извикване трябва да имат зададеното право.
- **Assert**, **Deny**, **PermitOnly** – управляват работата на "Stack Walk".

Например следният атрибут указва, че даденото асембли изисква задължително права за достъп до всички файлове на дисковото устройство C:

```
[assembly:FileIOPermission(
  SecurityAction.RequestMinimum, All="C:\\")]
```

Възможно е и да се укажат ограничения върху това кои асемблита могат да дефинират класове наследяващи определен клас. Следният атрибут на клас указва че класът може да бъде наследяван само в асемблита подписани с ключ удостоверен със сертификата **certificate.cer**:

```
[PublisherIdentityPermission(SecurityAction.InheritanceDemand,
  CertFile = "certificate.cer")]
public class SomeClass
{
  //...
}
```

Програмно искане на права

Програмното искане на права позволява на кода да иска права по време на изпълнението си. Това се осъществява чрез извикване на метода **Demand()** на обект от клас реализиращ права. Той проверява дали текущото асембли и всички извикващи го асемблита по стека имат поисканото право. Това предизвиква обхождането на стека (Stack Walk) и ако правото бъде отказано се предизвиква изключение.

Следният код проверява дали изпълняваният код притежава права за показване на диалог за избор на файл:

```
FileDialogPermission fdPerm = new FileDialogPermission(
  PermissionState.Unrestricted);
fdPerm.Demand();
```

Ако необходимите права не са налични, CLR хвърля **SecurityException** по време на изпълнение на приложението.

Сигурност базирана на роли (Role-Based Security)

В предишната секция разгледахме как чрез Code Access Security можем да управляваме правата на различните асемблита за достъп до извикваните от тях други асемблита. Сега ще разгледаме схемата, която .NET Framework предлага за управление на правата на изпълнение на базата на това в какви **роли** участва текущият потребител. Роля наричаме символно означение на категория потребители, които имат едни и същи привилегии, например: Guest, Administrator, Manager и т.н.

Role-Based Security е схема, чрез която можем да запазваме информация за самоличността на потребителя и асоциираните с него роли и в последствие да проверяваме какви права има той (оторизация). Целта е чрез парола, сертификат, смарт-карта или друг метод да установим дали потребителят е този, за когото се представя. След като знаем кой е текущият потребител, можем при всеки опит за достъп до даден клас да извършваме проверка в предварително дефиниран набор от правила дали потребителят има право на този достъп.

Автентикация и оторизация

Преди да преминем по-нататък, нека обясним в детайли какво означават термините "автентикация" и "оторизация".

Автентикация (authentication) е процесът на проверка дали даден потребител е този, за който се представя. Може да се извършва с парола, с цифров сертификат, със смарт-карта или по друг начин.

Оторизация (authorization) е процесът на проверка дали даден потребител има право да извърши дадено действие (предполага се, че потребителят е успешно автентикиран). Role-Based Security осигурява механизми за оторизация в .NET приложенията.

Класовете Identity и Principal

За извършване на оторизация чрез Role-Based Security се използват класовете **`Identity`** и **`Principal`**. Класът **`Identity`** носи информация за потребителя, в чийто контекст се изпълнява кода. В него се съхранява потребителско име и в зависимост от типа Identity може да се пази име на домейн, дали потребителят е автентикиран и др. Класът **`Principal`** представлява колекция от роли. Чрез изброените роли в един обект **`Principal`** можем да проверяваме какви права са отредени на потребителя.

В .NET Framework има два типа **`Identity`** и **`Principal`** класове:

- **`WindowsIdentity`** и **`WindowsPrincipal`**
- **`GenericIdentity`** и **`GenericPrincipal`**

Работа с WindowsIdentity и WindowsPrincipal

WindowsIdentity и **WindowsPrincipal** представляват потребителите и техните роли в контекста на Microsoft Windows. Те съдържат специфична за тази операционна система информация и употребата им е уместна, ако наборът от роли в Windows е подходящ за целите на приложението.

Ето два примера за създаване на **WindowsIdentity** и **WindowsPrincipal** обекти:

```
WindowsIdentity winIdentity = WindowsIdentity.GetCurrent();
Console.WriteLine("Windows login: {0}", winIdentity.Name);

WindowsPrincipal winPrincipal =
   new WindowsPrincipal(winIdentity);
```

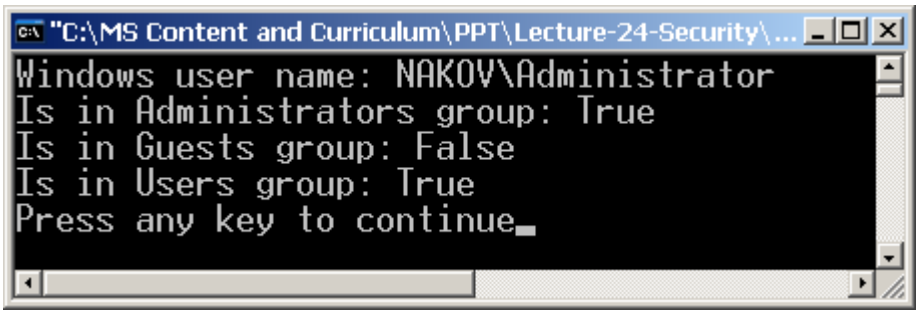
С горния блок инициализираме нов обект от тип **WindowsIdentity** със статичния метод **WindowsIdentity.GetCurrent()** и показваме името на потребителя на екрана.

Информация за текущия потребител – пример

С настоящия пример ще илюстрираме как от .NET Framework може да се извлече информация за текущия Windows потребител, под който е стартирано приложението:

```
WindowsIdentity winIdentity = WindowsIdentity.GetCurrent();
Console.WriteLine("Windows user name: {0}", winIdentity.Name);

WindowsPrincipal winPrincipal = new
   WindowsPrincipal(winIdentity);

bool isAdmin =
   winPrincipal.IsInRole(WindowsBuiltInRole.Administrator);
Console.WriteLine("Is in Administrators group: {0}", isAdmin);

bool isGuest = winPrincipal.IsInRole(WindowsBuiltInRole.Guest);
Console.WriteLine("Is in Guests group: {0}", isGuest);

bool isUser = winPrincipal.IsInRole(WindowsBuiltInRole.User);
Console.WriteLine("Is in Users group: {0}", isUser);
```

При изпълнение на примера се получава следният резултат:



Работа с GenericIdentity и GenericPrincipal

Когато използваме **GenericPrincipal**, за да изградим собствена схема за автентикация и оторизация, трябва да се придържаме към следния план:

1. Автентикация на потребителя

   При стартирането на приложението или при обръщение към даден ресурс, предназначен само за потребители с определени права, изискваме от потребителя да въведе потребителско име и парола. Извършваме проверка дали потребителското име и паролата са валидни.

```
if (ValidLogin(user, pass))
{
  // User authenticated
}
```

2. Създаване на **GenericIdentity** и **GenericPrincipal** обекти

   След като знаем, че имаме валиден потребител, създаваме за него **GenericIdentity** и **GenericPrincipal** обекти. **GenericIdentity** инициализираме с потребителското име, а **GenericPrincipal** с новосъзданеното **GenericIdentity** и списък от ролите, в които участва потребителят.

```
GenericIdentity id = new GenericIdentity("some user");
string[] roles = {"Manager", "Developer", "QA"};
GenericPrincipal prin = new GenericPrincipal(id, roles);
```

3. Асоцииране на **Principal** обекта с текущата нишка

   За да улесним последващи проверки, задаваме инициализирания текущо **GenericPrincipal** като текущ **Principal** на нишката.

```
System.Threading.Thread.CurrentPrincipal = prin;
```

Оторизация по Principal обект

След като сме изградили схема за асоцииране на потребителя с **Principal** обект, можем да ползваме тази информация, за да проверяваме дали той

има право да изпълни дадена част от кода. Тази проверка можем да правим по два начина – декларативно и програмно.

Декларативна оторизация

Декларативна оторизация правим чрез атрибути. Можем да ги задаваме на две нива – на ниво метод и на ниво клас. При задаването им на метод, проверката се извършва при извикване на метода, а ако са зададени на клас – при създаването на обект от този клас. Припомнете си, че единицата за задаване на права в Code Access Security беше асемблито.

Следните два примера илюстрират задаване на Role-Based Security атрибути на метод. В първият пример задаваме изискване потребителят да участва в ролята "Developer", а във втория - името му да е "Иванов". И в двата случая, ако условието не е изпълнено, при извикването на метода се генерира изключение от тип **SecurityException**.

```
[PrincipalPermission(SecurityAction.Demand,
  Role="Developer", Authenticated=true)]
public void DoSomething()
{
  // Perform some action
}

[PrincipalPermission(SecurityAction.Demand, Name="Иванов")]
public void DoSomethingElse()
{
  // Perform some action
}
```

В имената на потребителите и ролите не се прави разлика между малки и главни букви – "developer" и "DevelopeR" са равнозначни.

Ако имаме няколко потребителя в различни роли, на които трябва да бъде позволен достъп, можем да зададем повече от един атрибут. Ако текущият потребител изпълнява условието, зададено в поне един от атрибутите, на него ще му бъде позволен достъп.

```
[PrincipalPermission(SecurityAction.Demand, Role="Teller")]
[PrincipalPermission(SecurityAction.Demand, Role="Manager")]
public class Statement
{
  // Class for account statement
}
```

Ако потребителят има ролята "Teller" и/или ролята "Manager" той може да създава обекти от този клас. В противен случай при опит за създаване на обект ще се генерира изключение от тип **SecurityException**.

За да правим разлика в рамките на един метод какви права имат потребителите от различните роли, трябва да прибегнем до програмна оторизация.

Програмна оторизация

Освен чрез атрибути, можем да проверяваме правата на потребителя и програмно. Въпреки че атрибута **PrincipalPermission** е много удобен, в някои случаи може да се наложи да ползваме програмна проверка. Пример за такъв случай е ако решение дали да бъде позволен достъп зависи освен от ролята на потребителя и от друга стойност, която не е известна преди изпълнението на програмата.

Ето пример за програмна проверка по роля:

```
if (principal.IsInRole("Administrators"))
{
   // Perform some action
}
```

Проверка по потребителско име:

```
if (principal.Identity.Name == "Пешо")
{
   // Perform some action
}
```

Проверка чрез създаване на нов **PrincipalPermission** обект:

```
PrincipalPermission prinPerm = new
   PrincipalPermission("Пешо", "Tester");
prinPerm.Demand();
// Throws SecurityException if the check fails
```

В горния пример създаваме инстанция на **PrincipalPermission** и в конструктора му задаваме условията, които се изискват от потребителя. При извикване на метода **Demand()** обекта от тип **PrincipalPermission** извършва сравнение на подадените му потребителско име и роля с текущите за нишката. Ако сравнението е неуспешно се генерира изключение от тип **SecurityException**.

Можем да проверяваме само ролята на потребителя, като за целта подаваме **null** като параметър за потребителско име.

Оторизация с потребители и роли – пример

Със следващия пример ще илюстрираме възможностите, които .NET Framework предлага, за оторизация, базирана на потребители и роли:

```
using System;
```

```
using System.Security.Principal;
using System.Security.Permissions;

class RoleBasedSecurityDemo
{
   static void Main()
   {
      Console.Write("Username: ");
      string user = Console.ReadLine();
      Console.Write("Password: ");
      string pass = Console.ReadLine();

      if (ValidLogin(user, pass))
      {
         // Create generic identity and principal objects
         GenericIdentity identity = new GenericIdentity(user);
         string[] roles = {"Manager", "Developer", "QA"};
         GenericPrincipal principal =
            new GenericPrincipal(identity, roles);

         // Attach the principal to the current thread
         System.Threading.Thread.CurrentPrincipal = principal;

         DoSecuredOpearation();
      }
      else
      {
         Console.WriteLine("Invalid login.");
      }
   }

   static bool ValidLogin(string aUsername, string aPassword)
   {
      bool valid = (aUsername == aPassword);
      return valid;
   }

   [PrincipalPermission(SecurityAction.Demand, Name="Admin")]
   static void DoSecuredOpearation()
   {
      Console.WriteLine("Secure opration invoked.");
      IPrincipal principal =
         System.Threading.Thread.CurrentPrincipal;
      Console.WriteLine("User: {0}", principal.Identity.Name);

      bool isManager = principal.IsInRole("Manager");
      Console.WriteLine("Is Manager: {0}", isManager);

      bool isGod = principal.IsInRole("God");
      Console.WriteLine("Is God: {0}", isGod);
```

```
    }
}
```

Как работи примерът?

При стартиране приложението иска от потребителя да въведе име и парола, след което ги проверява (извършва автентикация). Автентикацията може да се извърши по множество начини (например чрез проверка в база данни или в LDAP директория), но за целите на примера просто се проверява дали потребителското име съвпада с паролата.

При успешна автентикация се създава обект **`GenericIdentity`**, в който се записва името на потребителя. След това на потребителя се задават роли и се създава **`GenericPrincipal`** обект. Този **`GenericPrincipal`** обект се асоциира с текущата нишка, което позволява след това по него да се извършват декларативни и програмни проверки на потребителя и ролята.

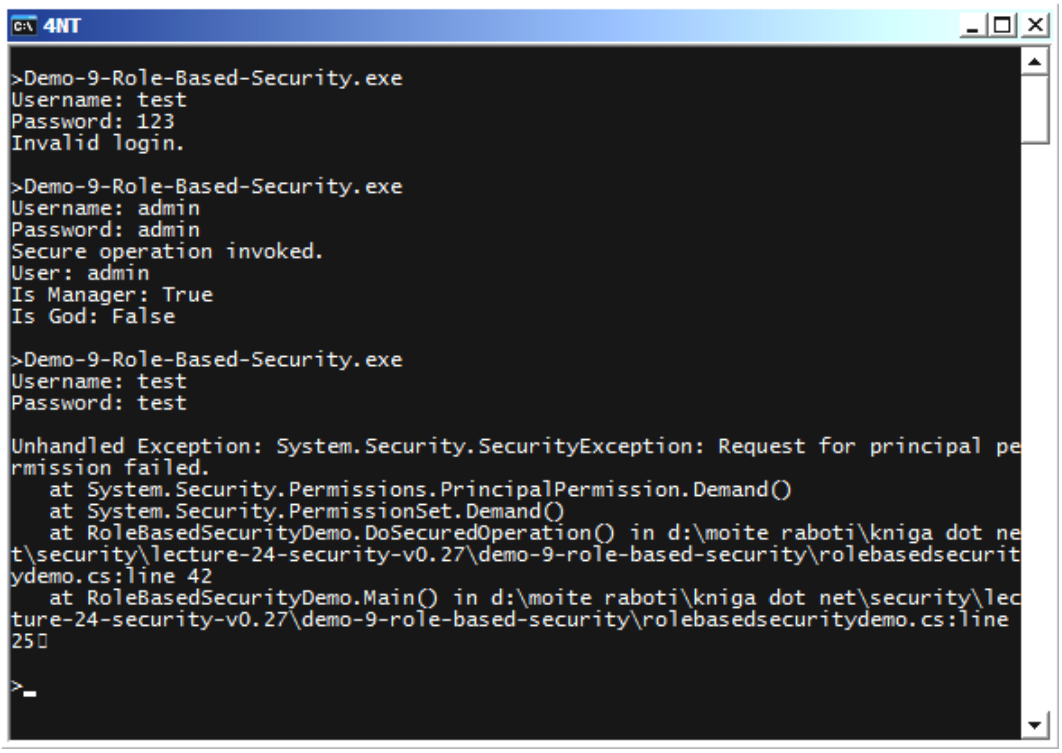
В примера е реализиран метод **`DoSecuredOpearation()`**, на който декларативно е указано, че изисква потребителят, асоцииран с текущата нишка, да е с име "**`Admin`**". Ако при извикване на метода потребителят е друг, CLR ще генерира изключение от тип **`SecurityException`**.

Примерът в действие

За да илюстрираме приложението, ще го стартираме и ще изпълним следните стъпки:

1. Въвеждаме потребител и парола, които не съвпадат, и виждаме, че автентикацията на потребителя не успява.
2. Въвеждаме потребител "admin" и парола "admin" и виждаме, че автентикацията успява и методът **`DoSecuredOperation()`** се извиква успешно.
3. Въвеждаме потребител "test" и парола "test" и виждаме, че автентикацията успява, но при опит за извикване на метода **`DoSecuredOperation()`**, CLR генерира изключение от тип **`SecurityException`**, защото текущият потребител не е "Admin".

Резултатът изглежда по следния начин:



Криптография в .NET Framework

.NET Framework предлага богат набор от средства за работа с криптографски алгоритми. Класовете в пространството **`System.Security.Cryptography`** позволяват работа с алгоритми за извличане на хеш стойност, симетрични криптиращи алгоритми, асиметрични криптиращи алгоритми, цифрови подписи и сертификати.

Извличане на хеш стойност

В началото на настоящата тема обяснихме, че хеширането е процес на "смилане" на даден документ, при който от него се извлича кратка последователност от байтове, наречена хеш-стойност. Сега ще се спрем на средствата, които .NET Framework ни дава, за работа с хеширащи алгоритми.

В .NET Framework има имплементирани класове за извличане на хеш стойности по стандартите MD5, SHA1, SHA256, SHA384 и SHA512. За извличане на хеш с ключ са имплементирани класовете **`HMACSHA`** и **`MACTripleDES`**, които ползват съответно алгоритмите SHA-1 и 3DES.

Изчисляването на хеш стойност за дадена поредица от байтове става с едно обръщение към метода **`ComputeHash()`** на съответния клас.

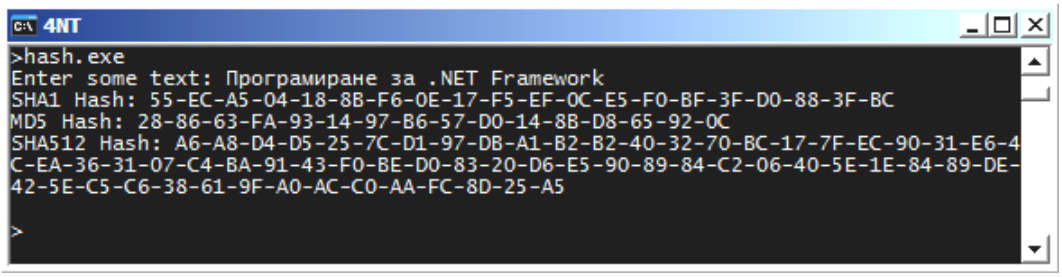
Извличане на хеш – пример

Със следващия пример ще илюстрираме как можем да изчислим SHA-1 хеш от дадено текстово съобщение:

```
using System;
using System.Security.Cryptography;
using System.Text;

class HashSample
{
   static void Main()
   {
      Console.Write("Enter some text: ");
      string s = Console.ReadLine();
      byte[] data = Encoding.ASCII.GetBytes(s);

      SHA1CryptoServiceProvider sha1 =
         new SHA1CryptoServiceProvider();
      byte[] sha1hash = sha1.ComputeHash(data);
      Console.WriteLine("SHA1 Hash: {0}",
         BitConverter.ToString(sha1hash));

      MD5CryptoServiceProvider md5 =
         new MD5CryptoServiceProvider();
      byte[] md5hash = md5.ComputeHash(data);
      Console.WriteLine("MD5 Hash: {0}",
         BitConverter.ToString(md5hash));

      SHA512 sha512 = new SHA512Managed();
      byte[] sha512hash = sha512.ComputeHash(data);
      Console.WriteLine("SHA512 Hash: {0}",
         BitConverter.ToString(sha512hash));
   }
}
```

За да изчислим хеш стойността на една байтова поредица създаваме обект, съответстващ на желаният тип хеш, и я подаваме като параметър на метода му **`ComputeHash()`**.

Резултатът от изпълнението на примера е следният:



Симетрични криптиращи схеми

В началото на темата вече разгледахме симетричните кодиращи схеми и обяснихме, че при тях се използва един и същ ключ за криптиране и декриптиране на информацията. Нека сега разгледаме какви средства ни предоставя .NET Framework за извършване на криптиране и декриптиране със симетрични криптографски алгоритми.

Криптиране и декриптиране с CryptoStream

За поточно симетрично криптиране и декриптиране в .NET Framework е имплементиран класът **`CryptoStream`**. Нека разгледаме схемата на работа.

За да криптираме или декриптираме даден текст първо създаваме обект от типа на избрания алгоритъм. Например за Rijndael/AES класът е **`RijndaelManaged`**, а за 3DES – **`TripleDESCryptoServiceProvider`**. Задаваме като параметри ключа и **началния вектор**. Началният вектор е необходим за всички алгоритми, които разделят текста на блокове и използват резултата от криптиране на предходния блок при криптирането на текущия. Началният вектор се използва при криптирането на първия блок, тъй като за него няма предходен. При декриптирането трябва да се използва освен същата парола и същия начален вектор.

Следващата стъпка е да създадем **`CryptoStream`** обект. При инициализирането му задаваме обекта с избрания алгоритъм, и обект от тип **`Stream`**, където да се съхранява резултата.

Криптирането се осъществява като пишем в **`CryptoStream`** потока и четем в подадения му като параметър изходен поток, който може да е файл, низ, мрежа и т.н.

Криптиране с CryptoStream – пример

Следният пример показва как можем да криптираме Unicode текст:

```
using System.Security.Cryptography;
using System.IO;
using System.Text;
...
// Instantiate cryptographic scheme
Rijndael cryptoAlg = new RijndaelManaged();

// Get random bytes (salt) to help generate secure password
RandomNumberGenerator randNum = new RNGCryptoServiceProvider();
byte[] salt = new byte[32];
randNum.GetBytes(salt);

// Generate the password
PasswordDeriveBytes passProvider = new
   PasswordDeriveBytes("моята парола 213", salt);
byte[] password = passProvider.GetBytes(32);
```

```
// Initialize the algorithm object with the password and IV
cryptoAlg.Key = password;
cryptoAlg.GenerateIV();

// Create a stream destination for the encryption
MemoryStream msDestination = new MemoryStream();

// Create the CryptoStream and set the msDestination
// memory stream as its target
CryptoStream csEncryptor = new CryptoStream(
  msDestination, cryptoAlg.CreateEncryptor(),
  CryptoStreamMode.Write);

// Transform the input text as byte sequence and
// write it to the stream
byte[] byteInput =
  Encoding.Unicode.GetBytes("моята тайна информация");
csEncryptor.Write(byteInput, 0, byteInput.Length);
csEncryptor.FlushFinalBlock();

// CryptoStream csEncryptor has encrypted the data
// into the stream. Retrieve it:
encrypted = msDestination.ToArray();
// Result: encrypted[] byte array has the encrypted text
```

В показания пример използваме Rijndael/AES за криптиране на текста. За получаване на ключа за криптиране използваме както потребителската парола така и случайна стойност, наречена "**сол**". Солта е стойност, която се използва обикновено при алгоритмите за хеширане, за да се получават различни хеш стойности за една и съща входна стойност (например парола). В зависимост от конкретното решение, тази стойност може или да бъде пазена в тайна или да е публично известна. Дори да е публично известна, тя затруднява така наречените речникови атаки, тъй като на атакуващия паролата се налага да преизчислява хеш стойностите с дадената сол (не може да ги има на готово).

За да е наистина произволна солта, използваме предоставения от .NET Framework генератор на случайни числа **`RandomNumberGenerator`**. Той е криптографски силен генератор на случайни числа.

За получаване на произволен начален вектор използваме метода на **`RijndaelManaged GenerateIV()`**. Дори да не извикаме този метод, при инициализацията на всеки симетричен алгоритъм за криптиране се генерират произволен ключ и начален вектор. Необходимо е да запазим началният вектор, тъй като декриптирането трябва да се извърши със същия вектор.

Създаваме **`MemoryStream`** обект, в който **`CryptoStream`** ще подава вече криптирания текст. На **`CryptoStream`** задаваме чрез първият параметър, че ще работи с новосъздадения **`MemoryStream`** обект и чрез третият параме-

тър, че **CryptoStream** ще пише в него. Това е необходимо, тъй като както ще видим по-нататък, при декриптирането **CryptoStream** ще чете от подадения му като параметър поток. С втория параметър ние създаваме самия обект, който ще извършва криптирането. Така **CryptoStream** ще трансформира всяка стойност, която напишем в него чрез този обект, и ще изпраща изхода в зададения от нас поток.

Както се вижда от примера, криптиращият обект очаква входът да е в байтова поредица. За да получим тази байтова поредица използваме метода **GetBytes()**, който разгледахме в темата за низовете.

Тъй като Rijndael/AES алгоритъмът работи с блокове, необходимо е след като сме написали последната част от входния текст да накараме **CryptoStream** да подаде оставащите символи, дори да не са с размер достатъчен за един блок, и след като се допълнят със символи до дължината на блока да се криптират и резултатът да се отрази на изходния поток. Това правим с метода **FlushFinalBlock()**.

Полученият резултат е байтовата поредица **encrypted**, която съдържа криптираната версия на низа.

Декриптиране с CryptoStream – пример

Ето как можем да декриптираме съобщението получено при работата на предходния пример:

```
using System.Security.Cryptography;
using System.IO;
using System.Text;
...
// Instantiate cryptographic scheme
Rijndael cryptoAlg = new RijndaelManaged();

// Load the encrypted data in byte array encrypted[]
// ...

// Load the password and initialization vector
// ...

// Create the stream used as source for decryption
MemoryStream msSource = new MemoryStream(encrypted);

// Set cryptoAlg with password and IV used during encryption
cryptoAlg.Key = password;
cryptoAlg.IV = IV;

// Create a cryptographic stream
CryptoStream csDecryptor = new CryptoStream(msSource,
  cryptoAlg.CreateDecryptor(), CryptoStreamMode.Read);
// Create buffer to store the decrypted data
byte[] result = new byte[encrypted.Length];
```

```
// Read the decrypted data
csDecryptor.Read(result, 0, result.Length);

// Get the resulting string
string resultString = ASCIIEncoding.Unicode.GetString(result);
```

Създаваме обекта криптографски алгоритъм Rijndael и му задаваме паролата и началния вектор, които използвахме при криптирането. Създаваме **MemoryStream** поток, чрез който да се осъществява достъп до криптираното съобщение. Задаваме го на **CryptoStream** потока, като чрез параметъра **CryptoStreamMode.Read** му указваме да чете от него.

Така когато четем от **CryptoStream** потока той взима входното съобщение и чрез **cryptoAlg** обекта го декриптира. Прочетената байтова поредица **result** преобразуваме в Unicode низ, който съдържа оригиналното съобщение.

Асиметрични криптиращи схеми

В началото на настоящата тема разгледахме какво представляват асиметричните криптиращи схеми и обяснихме, че те използват двойка криптографски свързани ключове (публичен и личен) и, че кодираните с единия ключ данни могат да се декодират само с другия. Сега ще разгледаме в детайли как можем да използваме асиметрични кодиращи алгоритми.

В .NET Framework са имплементирани алгоритмите RSA и DSA чрез класовете **RSACryptoServiceProvider** и **DSACryptoServiceProvider**. Тъй като алгоритмите за асиметрично криптиране са много по-бавни от тези за симетрично, не е предвидено те да работят с **CryptoStream**.

Инстанциране на доставчик за асиметрично криптиране

Нека видим как се инстанцира доставчик за асиметрично криптиране и декриптиране:

```
// Instantiate asymmetric encryption provider
RSACryptoServiceProvider rsaProvider = new
   RSACryptoServiceProvider();
```

Със създаването на **RSACryptoServiceProvider** автоматично се генерират двойка ключове и начален вектор. В конструктора не сме задали дължината на ключа и затова се генерира ключ с дължината по подразбиране - 1024 бита.

За RSA ключовете

Минимална дължина на RSA ключ, който можем да зададем, е 384 бита, а максималната – 16384. Стъпката между позволените стойности е 8 бита.

Дължината на ключа определя максималния размер на байтовата поредица, която може да бъде криптирана. За да определим този размер трябва да извадим числото 11 от размера на ключа в байтове.

Така например с ключ с дължина от 128 байта (1024 бита) можем да криптираме 128 – 11 = 117 байта, а ако дължината на ключа е 256 байта (2048 бита) максимумът е 256 – 11 = 245 байта.

Според спецификациите на RSA Labs, за да е сигурен ключът поне до 2010 година, той трябва да е с минимална дължина от 1024 бита, поне 2048 бита дават сигурност до 2030 година, а 3072 бита дължина е минимумът за ключ, който да е сигурен след 2030 година. Тези стойности са приблизителни и важат единствено, ако компютърната производителност продължава да се увеличава по закона на Мур.

Правилото "по-дългият ключ е по-добър" не винаги е добър водач, тъй като безразсъдно големите ключове забавят в голяма степен обработката на данните и изискват прекомерно големи ресурси. При изграждането на всяка система трябва да се направи анализ какъв вариант е най-добър.

Извличане на ключове

За да получим стойностите на генерираните ключове можем да ползваме метода **`ExportParameters(bool)`**. Чрез булев параметър задаваме дали да се извлече и личният ключ или само публичният.

```
RSACryptoServiceProvider rsaProvider1 =
   new RSACryptoServiceProvider();

// Export keys and store them in RSAParameters
RSAParameters bothKeys = rsaProvider.ExportParameters(true);
RSAParameters publicKey = rsaProvider.ExportParameters(false);

RSACryptoServiceProvider rsaProvider2 =
   new RSACryptoServiceProvider();
// Import keys to another RSACryptoServiceProvider
rsaProvider2.ImportParameters(publicKey);
```

В горния пример извличаме публичния ключ и го задаваме на друг криптиращ обект. За съхраняване на стойността на ключа използваме обект от тип **`RSAParameters`**.

Друго полезно средство е възможността да се извличат публичния ключ/двойката ключове в XML низ, за взаимодействие с други приложения. Ето как изглежда полученият XML низ за най-късия възможен RSA ключ – 384 бита:

```
// Export keys to XML
RSACryptoServiceProvider rsaEncoder = new
   RSACryptoServiceProvider(384);
string keys = rsaEncoder.ToXmlString(true);
```

Ето и съдържанието на променливата **keys**:

```
<RSAKeyValue>
  <Modulus>4odc9GTIkS1W1X94pE/ythvB6ASZsU2f5z
  8xOLxhoOzjaJZPgG+LrRzoxIrjV0NP</Modulus>
  <Exponent>AQAB</Exponent>
  <P>86S1l84iiIxqW8pi1G1JtJnVszKVPeEL</P>
  <Q>7gRzHNNiKtQvvb619I9Z7tR9RzLHkZlN</Q>
  <DP>uvuVQcO5TQI2Peu8nTqibjABiV0wnCSx</DP>
  <DQ>INTcTA2cbOv36eR0lNdxQFBvN3L5tEvB</DQ>
  <InverseQ>tFfBlzDmvIdgT6BDavVTLkwZb8bZvOHE</InverseQ>
  <D>RKq3uLWcPrW5rroXPeemMSG047oRRLe8gQD7z9+8
  vJ1b04Sz42QHhnfPnBlH8H0x</D>
</RSAKeyValue>
```

Извличаме стойността на най-късия възможен ключ с цел прегледност. Представянето е винаги в същия формат, променя се единствено дължината.

Криптиране и декриптиране на съобщение

Нека видим пример за криптиране на съобщение и декриптирането му. Нека кръстим изпращача Асен а получателят – Борис. Асен има XML файл **borisPBK.xml**, съдържащ публичния ключ на Борис. Борис има своята двойка публичен/личен ключ във файла **borisKeyPair.xml**.

Ето как Асен може да създаде тайно съобщение за Борис:

```
// Instantiate asymmetric encryption provider
RSACryptoServiceProvider rsaProvider = new
  RSACryptoServiceProvider();

// Convert input to byte array
byte[] toEncrypt = Encoding.Unicode.GetBytes("тайно съобщение");

// Load Boris' public key
XmlDocument xmlPBKey = new XmlDocument();
xmlPBKey.Load("borisPBK.xml");
rsaProvider.FromXmlString(xmlPBKey.InnerXml);

// Call Encrypt() method to do actual encryption
byte[] encrypted = rsaProvider.Encrypt(toEncrypt, false);

// Save the encrypted message
FileStream fs = new FileStream("encrypted_message.txt",
  FileMode.Create);
using (BinaryWriter bw = new BinaryWriter(fs))
{
  bw.Write(encrypted.Length);
  bw.Write(encrypted);
}
```

```
// Result: encrypted_message.txt contains encrypted message
```

Тъй като съобщението е за Борис, трябва да използваме неговия публичен ключ. За да прочетем ключа от XML файла създаваме **XmlDocument** и зареждаме в него XML представянето на публичния ключ. След това го задаваме на RSA доставчика **rsaProvider** и криптираме съобщението. Получената байтова поредица съхраняваме във файла **encrypted_message.txt**, като първо записваме нейната дължина.

След като получи съобщението, Борис би могъл да го провери по следния начин:

```
// Instantiate new asymmetric encryption provider
RSACryptoServiceProvider rsaProvider = new
   RSACryptoServiceProvider();

// Load Boris' private key (key pair)
XmlDocument xmlPKey = new XmlDocument();
xmlPKey.Load("borisKeyPair.xml");
rsaProvider.FromXmlString(xmlPKey.InnerXml);

// Load the encrypted message
byte[] encrypted;
FileStream fs = new FileStream("encrypted_message.txt",
   FileMode.Open);
using (BinaryReader br = new BinaryReader(fs))
{
   int msgLenght = br.ReadInt32();
   encrypted = new byte[msgLenght];
   encrypted = br.ReadBytes(msgLenght);
}

// Call Decrypt() to get the decrypted byte array
byte[] decrypted = rsaProvider.Decrypt(encrypted, false);

// Convert byte array to Unicode character array
string sMessage = Encoding.Unicode.GetString(decrypted,
   0, decrypted.Length);
// Result: sMessage contains "тайно съобщение"
```

Борис задава двойката си ключове на RSA доставчика **rsaProvider**, за да бъде извършено декриптирането с неговия личен ключ. След това възстановява изпратената байтова поредица и извиква метода **Decrypt()**, за да получи оригиналното съобщение.

Работа с цифрови подписи

В .NET Framework са имплементирани два алгоритъма за цифров подпис: RSA и DSA. За цифрово подписване можем да използваме и XML подписи, които ще разгледаме в следващата секция.

RSA алгоритъмът се прилага както за асиметрично криптиране на данни, така и за създаване на цифрови подписи. DSA (Digital Signature Algorithm) се използва само за създаване на цифрови подписи и е създаден с идеята да стане стандарт за подписване.

Класовете, които имплементират тези два алгоритъма, са съответно **RSACryptoServiceProvider** и **DSACryptoServiceProvider**. И двата класа предоставят за създаване на подпис методи **SignData()** и **SignHash()** и за проверка на подпис методи **VerifyData()** и **VerifyHash()**.

Подписване на документ – пример

Ето как можем да подпишем тайно съобщение чрез алгоритъма RSA:

```
using System.Security.Cryptography;
using System.Text;
...
// Instantiate provider and generate random key pair
RSACryptoServiceProvider rsaProvider = new
  RSACryptoServiceProvider();
byte[] dataToSign =
  Encoding.Unicode.GetBytes("съобщение за подписване ...");

HashAlgorithm hashAlg = HashAlgorithm.Create("MD5");
byte[] result = rsaProvider.SignData(
  dataToSign, 0, dataToSign.Length, hashAlg);
// Result: result[] contains the signature
```

В примера създаваме обект от тип **RSACryptoServiceProvider**, по същия начин като в примера за асиметрична криптография в предишната секция. Преобразуваме съобщението, което ще подписваме, в байтова поредица, и създаваме обект за хеш алгоритъма. Този обект ще бъде използван от **rsaProvider** за създаване на хеш стойността, която после ще бъде криптирана с личния ключ. Методът **SignData()** връща цифровият подпис като байтова поредица.

Отново при инициализацията на обекта от тип **RSACryptoServiceProvider** се генерира двойка ключове. В реално приложение трябва да зададем нашият личен ключ, за да бъде валиден подписът.

Подписване на хеш стойност на документ – пример

Вместо да задаваме документа и алгоритъм за хеширане, можем сами да получим хеш кода на съобщението и да го подпишем. Ето как можем да подпишем хеш стойността на едно съобщение (например договор) чрез DSA алгоритъма:

```
using System.Security.Cryptography;
using System.Text;
...
```

```
// Instantiate provider and generate random key pair
DSACryptoServiceProvider dsaProvider = new
   DSACryptoServiceProvider();

byte[] msgBytes = Encoding.Unicode.GetBytes(
   "договор за подписване");

// Extract SHA1 hash
SHA1 shaHasher = new SHA1CryptoServiceProvider();
byte[] hashToSign = shaHasher.ComputeHash(msgBytes);

// Sign it
string hashName = CryptoConfig.MapNameToOID("SHA1");
byte[] digSig = dsaProvider.SignHash(hashToSign, hashName);

// Verify signature
bool isValid = dsaProvider.VerifyHash(
   hashToSign, hashName, digSig);
// Result: isValid is true
```

Първо създаваме **dsaProvider** и автоматично за нас се генерира двойка ключове с указаната дължина. **DSACryptoServiceProvider** работи с ключове с размер между 512 и 1024 бита и стъпка 64 бита.

След това вземаме хеш стойността на съобщението. DSA винаги работи с SHA1 хеш стойности. Въпреки че се очаква да подадем SHA1 хеш стойност, трябва да зададем името на хеширащия алгоритъм. Чрез метода **MapNameToOID(string)** от краткото име взимаме точното обозначение на алгоритъма. Например ако го извикаме с "SHA512", "SHA-512" или "**System.Security.Cryptography.SHA512**", винаги получаваме конкретната стойност, обозначаваща алгоритъма - "2.16.840.1.101.3.4.3".

На **SignHash()** подаваме като параметри хеш стойността на съобщението и точното обозначение на хеш алгоритъма. Върнатият резултат е поредица от байтове, съдържаща цифровия подпис.

Със съответния метод **VerifyHash()** проверяваме дали подписът е валиден. Тъй като използваме отново **dsaProvider**, който вече има генерирана двойка ключове, няма нужда да ги задаваме изрично. Ако използваме друг обект за проверката, трябва да му зададем публичния ключ на лицето, подписало съобщението чрез **ImportParameters(DSAParameters)** или **FromXMLString(string)**.

XML подписи

.NET Framework имплементира технологията XML-Signature (XMLDSIG) на W3C (World Wide Web Consortium) за подписване на XML документи. XML-Signature позволява подписване на XML документи и ресурси в Интернет, като подписът е или отделен XML документ или се обединява с подписвания обект в един XML документ.

Подобно на способите за подписване, които вече разгледахме, XML подписите доказват произхода и интегритета (липса на промени) на едно съобщение. Специфичното за тях е че са пригодени за работа с XML файлове и имат специални функции за работа с Интернет ресурси.

Едно от основните им качества е възможността за подписване на част от XML документ. Можем да подпишем само част от ресурсите в документа, независимо от типа им. Един подпис може да удостоверява поредица от символи, поредица от байтове и определени елементи от документа.

Това качество на XML подписите е много подходящо, когато например в даден бизнес процес се предава по верига XML документ, и на всеки етап от неговото допълване, лицето или отдела подписва само своите промени.

Структура на XML подписа

Структурата на XML-Signature подписа е комплексна и няма да се спираме подробно на нея. Ще разгледаме най-общия й вид, който е показан на фигурата. Знакът "**?**" означава нула или един елемент, "**+**" един или повече, а "*****" – нула или повече.

```
<Signature ID?>
  <SignedInfo>
    <CanonicalizationMethod/>
    <SignatureMethod/>
    (<Reference URI?>
      (<Transforms>)?
      <DigestMethod>
      <DigestValue>
    </Reference>)+
  </SignedInfo>
  <SignatureValue>
  (<KeyInfo>)?
  (<Object ID?>)*
</Signature>
```

Елементът **SignedInfo** указва какво е подписано и с кои алгоритми. В него се съдържа указател към ресурса/ресурсите, които са подписани, и информация за използвания алгоритъм за хеширане. **SignatureValue** е самият подпис в Base64 кодираща схема. **KeyInfo** е незадължителен елемент, който предоставя публичният ключ, чрез който да се провери подписа.Ако той липсва, трябва от контекста да е ясно какъв ключ да се използва. При схемата, в която подписът съдържа подписания обект, на мястото на **<Object ID>** се намира стойността на обекта. Както ще видим по-нататък, в .NET Framework има класове, съответстващи на повечето от елементите в подписа. Чрез свойствата на тези класове можем да контролираме директно съдържанието на подписания документ.

Видове XML-Signature подписи

Дефинираните в стандарта подписи са три вида. Когато целият подписван обект се намира в рамките на XML подписа, наричаме подписа **опаковащ** (enveloping signature). Когато подписът е поделемент в XML документа, го наричаме **опакован** (enveloped signature). Можем да използваме опакован XML подпис, за да подпишем част от или цял XML документ. Ако подписът е отделен XML документ това е така наречения **обособен подпис** (detached signature). Като такива можем да считаме и подписите, които разгледахме в предишната секция – RSA и DSA. При тях стойността на подписа е отделена от съобщението.

За да подпишем XML документ трябва да инстанцираме обект от тип **`SignedXml`**. Този обект представлява един подписан документ и отговаря на схемата показана по-горе. Чрез свойствата и методите си **`SignedXml`** ни позволява да работим с поделементите му, както ще видим в примерите.

Подписване и проверка на XML с опаковащ подпис – пример

Първо ще разгледаме подписване на XML файл с опаковащ подпис. Резултатът ще бъде нов XML файл, с главен елемент **`Signature`**, който ще съдържа цялата описателна информация за използваните алгоритми за хеширане и подписване и като свой поделемент ще съдържа оригиналния XML файл:

```
using System.Security.Cryptography;
using System.Security.Cryptography.Xml;
using Xml = System.Security.Cryptography.Xml;
using System.Xml;
...
// Load XML file to sign
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load("report.xml");

// Create name for the signed element and place it
// in a System.Security.Cryptography.Xml.DataObject
Xml.DataObject dataObj = new Xml.DataObject();
dataObj.Data = xmlDoc.ChildNodes;
dataObj.Id = "report";

// Identify what is to be signed
Reference reportRef = new Reference();
reportRef.Uri = "#report";

// Assign the object to sign and its
// identifier to the SignedXml object
SignedXml signedXml = new SignedXml();
signedXml.AddReference(reportRef);
signedXml.AddObject(dataObj);

// Create the signing algorithm and generate keys
```

```
DSA dsaProvider = new DSACryptoServiceProvider();

// Export key to XML file
// 'false' means export only the public key
string publicKey = dsaProvider.ToXmlString(false);
XmlDocument xmlKey = new XmlDocument();
xmlKey.LoadXml(publicKey);
xmlKey.Save("key.xml");

// Assign the algorithm and keys to be used
signedXml.SigningKey = dsaProvider;

// The SignedXml object does the signing
signedXml.ComputeSignature();

// Save the signature to file
XmlDocument signedXmlDoc = new XmlDocument();
signedXmlDoc.LoadXml(signedXml.GetXml().OuterXml);
signedXmlDoc.Save("xmlsig.xml");
```

Как работи примерът?

В дадения пример първо създаваме обект от тип **SignedXml**. След това зареждаме XML документа, който искаме да подпишем, в обект от тип **XmlDocument**. Това ни позволява на следващата стъпка да вземем списък с неговите елементи и да го зададем на **dataObj** – обект от тип **System.Security.Cryptography.Xml.DataObject**, който представлява частта от XML подписа, където се намира подписваният документ. Това важи единствено за опаковащия подпис, тъй като само при него подписваният документ се намира в рамките на Signature елемента. На свойството **Id** на **dataObject** задаваме име, което да характеризира подписваният документ.

Създаваме **Reference** обект, който ни дава достъп до Reference секцията в подписа. Задаваме вече избраното в **dataObj.Id** име, като прибавяме "**#**" отпред, за да обозначим, че наименованието се отнася до елемент в рамките на в съобщението, а не е външен идентификатор.

След като сме указали какво ще подписваме, задаваме алгоритъмът и ключът, които ще се използват. Инстанцираме обект от тип **DSACryptoServiceProvider()**, който при създаването си автоматично генерира двойка публичен/личен ключ. Публичният ключ ще ни трябва за да проверим в последствие подписа и затова го запазваме в XML файл. Тъй като това е рутинна операция, класът **DSA** има метод, който ни дава низ във формат XML с информация за ключа. Методът е **ToXmlString(bool)** и чрез единствения му параметър контролираме дали в низа се включва и личния ключ (параметър **true**) или само публичният (параметър **false**).

Свойството на **SignedXml**, на което трябва да зададем алгоритъма и ключа, е **SignedXml.SigningKey**.

След като сме задали какво и как ще подписваме извикваме методът на **SignedXml ComputeHash()**, който извършва самото подписване. Резултатът е в XML формат и можем да го извлечем чрез метода **GetXml()** на **SignedXml**.

Как изглежда подписаният XML?

Нека изпълним примера, за да видим как изглежда подписаният XML документ. Ето примерен входен XML документ, с който ще извършим пробно подписване:

report.xml

```
<report>
  <title>Report</title>
  <details>This is an important report</details>
</report>
```

Ето как би могъл да изглежда горният XML документ след подписването:

xmlsig.xml

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
    <CanonicalizationMethod
Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
    <SignatureMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1" />
    <Reference URI="#report">
      <DigestMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
      <DigestValue>iq1fiCvzg49hrUIiyy8ToAOsWE8=</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue>t7a9TUfN7N7uyFmriXsYPMNdhV0NTREXNc6thBu+9B7Jz
7z54mVckw==</SignatureValue>
  <Object Id="report">
    <report xmlns="">
      <title>Report</title>
      <details>This is an important report</details>
    </report>
  </Object>
</Signature>
```

При подписването се генерират и случайна двойка DSA публичен и личен ключ, като публичният се записва във файла **key.xml**. Ето как би могъл да изглежда този публичен ключ:

key.xml

```
<DSAKeyValue>
  <P>nbGhksQHc5XvaCFtRpRGYoJNZevu5UE4lgkYVAjC5H1Nedp7l4fpfxRPApg
L+ko7yOV9t52BjReJMqSlDmt+U7xQC83SmRiRby9yN7W2ngct/Z6Ut8Phi267RkT
kwjHkRUjxaoNuuX5sdc/L/Ah1SWuklv1PzN3SAwARojhqqD0=</P>
  <Q>y6qq3MzI07DY3q5+S2DNTpbZ2aE=</Q>
  <G>DaeLxKxnEli+ZID9V+7/Fk58ne3kSBkThA/k1o7AOSSLn9OiuMafjL9jk6L
r1Fov8evFF0JetIRWUf9JKi6azK9JdvJ97L0soPQilfakuLyzdtjXD9xHJ9RWkmd
8Lb2EogaLaiOgGYMXYjafIMGbxY1XWc9moUV+IKb8E68QaOQ=</G>
  <Y>QVLDSdSnDyyevvSZNJdr+fZF3IDPt1QJwzWPBdzA09pRp3VZeoRfLELWJYN
c0js+sM9BzMIjEiKuOovTZVsbhguSSulJQTOyhjhpdw1wq0duxq+RuoQB4DT1u7v
Mli2WWMb01QzfkB7x5y4/eWL748L+lkb75GmpYXayi/8i4vg=</Y>
  <J>xjbXeC84pQs57anel0CIN928jr+ffDXEQGz9kUo+csLB69DRCb5PAmm7prk
Xb2Qvu+Lah+yyXK/kh0Ov6BHS0MsrwqDFo7fLFJXzg4XMQjLtV/skeoTwDRO/p1R
2vCwDcpuE/1eESZmr2xa8</J>
  <Seed>Deui/8/vwZflDF9vnQ00Q8suJ5k=</Seed>
  <PgenCounter>AVw=</PgenCounter>
</DSAKeyValue>
```

Проверка на XML подпис – пример

Нека сега видим как можем да проверим сигнатурата на създадения по-горе подписан XML документ. За целта ни трябват подписания документ (той съдържа оригиналния като част от себе си) и публичния ключ, използван при подписването. Да разгледаме следния примерен код за проверка на XML подписа:

```
using System.Security.Cryptography;
using System.Security.Cryptography.Xml;
using System.Xml;
...
// Create object representing signed document
SignedXml signedXml = new SignedXml();

// Load the signature we saved in the previous example
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.PreserveWhitespace = false;
xmlDoc.Load("xmlsig.xml");

// Extract the signature as node collection
XmlNodeList sigNodeList =
   xmlDoc.GetElementsByTagName("Signature");
signedXml.LoadXml((XmlElement)sigNodeList[0]);

// Create instance of algorithm to verify signature
DSA dsaProvider = new DSACryptoServiceProvider();

// Load the public key we stored during signing
XmlDocument xmlKey = new XmlDocument();
```

```
xmlKey.Load("key.xml");

// Assign the original key to new instance of DSA
dsaProvider.FromXmlString(xmlKey.OuterXml);

// Create KeyInfo to store public key info
KeyInfo myKI = new KeyInfo();
// Get the public key value to use for verification
myKI.AddClause(new DSAKeyValue(dsaProvider));
signedXml.KeyInfo = myKI;

// Check the signature
bool isValid = signedXml.CheckSignature();
Console.WriteLine(isValid);
```

Как работи примерът?

За да извършим проверката отново създаваме обект от тип **SignedXml**. Посредством **XmlDocument** зареждаме в него частта Signature (която включва целият документ, тъй като използваме опаковащ подпис). На свойството **PreserveWhitespace** на **XmlDocument** обекта указваме **false**, за да бъдат игнорирани празните места в документа. Така дори между подписването и проверката да бъдат вмъкнати допълнителни празни места, това няма да попречи проверката да бъде успешна. Всяка друга промяна на подписания документ ще накара проверката да не успее.

Инстанцираме отново **DSACryptoServiceProvider** обект и му задаваме публичният ключ от двойката ключове, с чийто личен ключ подписахме съобщението. Това правим с метода на **DSA FromXmlString(string)**, който инициализира **DSACryptoServiceProvider** с параметрите, съдържащи се в подадения низ.

За да зададем ключа на **SignedXml**, създаваме обект от тип **KeyInfo**. Създаваме **DSAKeyValue** обект, който представлява поделемент на **DSA** и съдържа публичния ключ. Задаваме новополученият **DSAKeyValue** на **myKI** обекта. Сега **myKI** съдържа публичният ключ и сме готови да го зададем на **sXmlDoc**.

Извикаме методът **CheckSignature()** на **SignedXml**, за да се извърши проверка на валидността на подписа.

Подписване и проверка на XML с опакован подпис – пример

Нека разгледаме подписването на XML файл с опакован подпис:

```
using System.Security.Cryptography;
using System.Security.Cryptography.Xml;
using System.Xml;
...
// Load file to sign
```

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.PreserveWhitespace = false;
xmlDoc.Load("report.xml");

// Create SignedXml and pass it the XML to sign
SignedXml sXmlDoc = new SignedXml(xmlDoc);

// Create Reference to set signing options
Reference reportRef = new Reference();
reportRef.Uri = "";

// Specify canonical XML transformation without comments
Transform xmlTransform = new XmlDsigC14NTransform();
reportRef.AddTransform(xmlTransform);

// Specify transformation for enveloped signature
XmlDsigEnvelopedSignatureTransform env = new
   XmlDsigEnvelopedSignatureTransform();
reportRef.AddTransform(env);

// Pass the settings to SignedXml object
sXmlDoc.AddReference(reportRef);

// Create DSA asymmetric provider
DSA dsaProvider = new DSACryptoServiceProvider();

// Store the auto-generated key for later verification
XmlDocument xmlKey = new XmlDocument();
xmlKey.InnerXml = dsaProvider.ToXmlString(false);
xmlKey.Save("key.xml");

// Set signing options to signature object
sXmlDoc.SigningKey = dsaProvider;

// Perform signing
sXmlDoc.ComputeSignature();

// Get resulting signature
XmlElement xmlDigSig = sXmlDoc.GetXml();

// Add the signature to the original XML file
xmlDoc.DocumentElement.AppendChild(xmlDoc.ImportNode(
   xmlDigSig, true));

// Remove XML declaration, if any
if (xmlDoc.FirstChild is XmlDeclaration)
{
   xmlDoc.RemoveChild(xmlDoc.FirstChild);
}
```

```
// Store the resulting document
xmlDoc.Save("signed_report.xml");
```

Разликата спрямо предишния пример е, че тук указваме да се добавят два елемента в секцията трансформации на XML документа. Чрез добавянето на обект от тип **`XmlDsigC14NTransform`** задаваме трансформация на XML документа към канонична форма и задаваме трансформация за опакован подпис чрез обекта **`XmlDsigEnvelopedSignatureTransform`**. Тези трансформации ще се изпълнят преди да се извърши хеширането.

Отново записваме публичния ключ, за да можем да проверим подписа, и извикаме метода за пресмятане на подписа. Добавяме новополученият подпис, който е XML елемент, в оригиналния документ, и записваме новия документ в XML файл. Тъй като създаваме опакован подпис, подписът се намира в рамките на оригиналният XML документ.

Ето как можем да проверим дали опакования подпис, който създадохме, е валиден.

```
using System.Security.Cryptography;
using System.Security.Cryptography.Xml;
using System.Xml;
...
// Load XML document
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.PreserveWhitespace = false;
xmlDoc.Load("signed_report.xml");

// Load the signature element
SignedXml sXmlDoc = new SignedXml(xmlDoc);
XmlNodeList nodeList = xmlDoc.GetElementsByTagName("Signature");
sXmlDoc.LoadXml((XmlElement)nodeList[0]);

// Create DSA asymmetric provider
DSA dsaProvider = new DSACryptoServiceProvider();

// Load the public key we stored during signing
XmlDocument xmlKey = new XmlDocument();
xmlKey.Load("key.xml");
dsaProvider.FromXmlString(xmlKey.OuterXml);

// Create KeyInfo to store public key info
KeyInfo myKI = new KeyInfo();
// Get the public key value to use for verification
myKI.AddClause(new DSAKeyValue(dsaProvider));
sXmlDoc.KeyInfo = myKI;

// Verify signature
bool isValid = sXmlDoc.CheckSignature();
```

Чрез **XmlDocument** зареждаме подписания документ, който получихме по-горе. Частта Signature от XML файла (която обхваща целия документ) зареждаме в обекта **sXmlDoc**, с който пресъздаваме подписания документ.

Създаваме доставчик на асиметрично криптиране DSA и му задаваме публичния ключ, като го прочитаме от XML файла **key.xml**. Чрез обекта **myKI** от клас **KeyInfo** пресъздаваме секцията със стойността на подписа в подписания документ, и на нея задаваме ключа, използван от DSA при подписването.

Извикваме метода **CheckSignature()** на подписания документ **sXmlDoc**, и получаваме като резултат **true**.

Подписване и проверка на XML с обособен подпис – пример

Сега нека разгледаме как можем да подпишем ресурс в Интернет. Ще направим обособен подпис, като XML документа, който ще получим, ще съдържа URI идентификатор, показващ кой обект сме подписали, и стойността на подписа. Стойността на подписвания ресурс няма да се съдържа в XML документа.

Нека този път включим публичния ключ на подписващия в подписа. Така при проверката няма да се налага ключа да се зарежда изрично, но остава проблема да се гарантира, че публичният ключ е на изпращач, на когото имаме доверие.

Ето как подписваме ресурс по идентификатор (URI):

```
using System.Security.Cryptography;
using System.Security.Cryptography.Xml;
using System.Xml;
...
// Create object to hold XML signature
SignedXml xmlSigned = new SignedXml();

// Identify what is to be signed
Reference myUriRef = new Reference();
myUriRef.Uri = "http://localhost/my_page.html";

// Add the URI to the signature object
xmlSigned.AddReference(myUriRef);

// Create the signing algorithm and generate keys
DSA dsaProvider = new DSACryptoServiceProvider();

// Configure KeyInfo to store the public key
KeyInfo myKI = new KeyInfo();
myKI.AddClause(new DSAKeyValue(dsaProvider));
xmlSigned.KeyInfo = myKI;

// Configure key
```

```
xmlSigned.SigningKey = dsaProvider;

// Calculate signature
xmlSigned.ComputeSignature();

XmlDocument xmlSignature = new XmlDocument();
xmlSignature.LoadXml(xmlSigned.GetXml().OuterXml);
xmlSignature.Save("uri_signature.xml");
```

Този път на обекта Reference задаваме Интернет адрес в полето Uri. Отново инстанцираме DSA обект за подписването, но този път правим една допълнителна стъпка – добавяме публичния ключ в XML подписа. За целта създаваме KeyInfo обект, както правихме досега при проверките, и му задаваме публичният ключ генериран от **dsaProvider**. Добавяме го към **xmlSigned** за да го включим като част от подписа. Извикваме метода **ComputeSignature()** за да се изчисли подписа и го съхраняваме в XML файл.

Нека проверим дали подписът е валиден:

```
using System.Xml;
using System.Security.Cryptography.Xml;
...
// Load the document containing signature
XmlDocument signatureDoc = new XmlDocument();
signatureDoc.Load("uri_signature.xml");

// Set up an object to represent XML signature
SignedXml sXml = new SignedXml();
XmlNodeList signNodeList =
signatureDoc.GetElementsByTagName("Signature");
sXml.LoadXml((XmlElement)signNodeList[0]);

// No need to specify key, since we included it
bool isValid = sXml.CheckSignature();
```

В тази проверка извършваме само три стъпки – зареждаме XML подписа в обект **XmlDocument**, оттам взимаме елемента Signature и го зареждаме в обект **SignedXml**. В случая не се налага да зареждаме публичния ключ, тъй като го включихме в рамките на XML подписа. Когато извикаме **CheckSignature()** подписът се извлича от **KeyInfo** и се използва за проверката.

Упражнения

1. Опишете ключовите характеристики на сигурността в .NET Framework – безопасност на типовете, защита на паметта, защита от аритметични грешки, подписване на асемблитата, IsolatedStorage, Code Access Security, Role Based Security и др.

2. Напишете библиотека (Class Library проект във VS.NET), която съдържа клас със статичен метод PrintVersion(), който отпечатва на конзолата версията на асемблито, от което е зареден класа. Компилирайте асемблито в 2 различни версии (1.0 и 2.0), подпишете ги, направете ги със силни имена и ги инсталирайте в GAC. Реализирайте 2 конзолни приложения, които ползват съответно версия 1.0 и 2.0 на асемблито.

3. Напишете Windows Forms контрола за IE, която позволява създаване на албуми със снимки, които се съхраняват в IsolatedStorage за текущия потребител. Контролата трябва да позволява разглеждане на албума, добавяне и изтриване на снимки, които се съхраняват в IsolatedStorage.

4. Създайте Windows Forms контрола за IE, която може да отваря, редактира и записва текстови файлове на локалния диск на потребителя. По подразбиране отварянето на локален файл няма да работи. Направете асемблито на контролата да има силно име. Чрез Security Policy Editor дайте права за четене и писане на асемблито на контролата, като създадете Code Group по силното му име.

5. Напишете Windows Forms приложение, което позволява създаване и записване на текстови бележки. Приложението трябва да съхранява бележките във файл в профила на текущия потребител, ако има права за това или в IsolatedStorage ако няма права. Правата трябва да се проверяват програмно.

6. Напишете библиотека (DLL), която поддържа функционалност за регистриране на потребител по username и password и проверка на валидността на двойка username/password. Библиотеката трябва съхранява данните си в XML файл и да използва собствените си права за достъп до файла. Клиенти с ниски права, които не могат да четат файла, трябва да могат да ползват функционалността на библиотеката.

7. С помощта на Role Based Security направете приложение, което управлява потребителите в дадена система. Потребителите, техните пароли и ролите на всеки потребител трябва да се съхраняват в XML файл. Възможните роли за всеки потребител са Guest, User и Admin. Гостите в системата имат право да се регистрират и нищо друго. Потребителите в системата имат право да извличат списъка от всички регистрирани потребители. Администраторите имат право да редактират данните и ролите на всички потребители. При начално стартиране системата трябва да предлага форма за автентикация, която позволява влизане като някакъв потребител или влизане като гост без парола. Проверката на ролите да се реализира чрез GenericPrincipal.

8. Реализирайте приложението от предходната задача, като съхранявате паролите на потребителите не като чист текст, а като SHA1 хеш стойност. Дава ли това по-голяма сигурност за системата?

Използвана литература

1. Светлин Наков, Сигурност в .NET Framework – http://www.nakov.com/dotnet/lectures/Lecture-24-Security-v1.0.ppt
2. MSDN Lectures, Implementing Application Security Using the Microsoft .NET Framework – http://downloads.microsoft.co.za/MSDNEssentials/20040402/AppSecurity.ppt
3. Бизнес в Интернет, Глава 7, Сигурност в Интернет – http://www-it.fmi.uni-sofia.bg/courses/BonI/chapter7.html
4. Derek Simon, Strong-Named Assemblies – http://www.incandesoft.com/development/strong-named%20assemblies.pdf
5. Chris Tavares, Understanding Isolated Storage – http://www.dotnetdevs.com/articles/IsolatedStorage.aspx
6. Adam Freeman & Allen Jones, Programming .NET Security, O'Reilly, 2003, ISBN 0-596-00442-7

Глава 27. Mono – свободна имплементация на .NET Framework

Автори

Антон Андреев

Цветелин Андреев

Необходими знания

- Базови познания за .NET Framework и CLR (Common Language Runtime)
- Базови познания за UNIX

Съдържание

В тази тема...

В настоящата тема ще разгледаме една от алтернативите на Microsoft .NET Framework – проектът с отворен код Mono. Ще обясним накратко начините за инсталиране и работа с Mono. Ще се запознаем с неговите компоненти: компилатори, виртуални машини, дебъгер, дизасемблер и др. Ще обърнем внимание на особеностите при използване на ASP.NET уеб приложения и уеб услуги върху сървърите Apache с mod_mono и XSP. Ще разгледаме още достъпът през ADO.NET до MySQL, PostgreSQL и други сървъри за бази от данни. Ще направим преглед на средствата за създаване на графични приложения: Windows Forms, Glade#, Gtk# и др. Ще дадем и няколко съвета относно писането на преносим код (такъв, който можете да компилирате на различните операционни системи без промени). Ще ви запознаем накратко и с работа с графика под Mono.

Проектът Mono

Проектът Mono (www.mono-project.com) е инициатива, която има за цел да реализира свободна версия на .NET Framework за Linux, Solaris, Mac OS X, Windows и други UNIX-базирани операционни системи. Спонсориран е от Novell, голяма ИТ компания, водещ доставчик на операционни системи, мрежов и системен софтуер. Mono е продукт с изцяло отворен код, базиран на ECMA/ISO стандартите. Разпространява се свободно под лицензи GNU/GPL и MIT X11.

Значение на проекта

Приложенията в .NET Framework се компилират до език от по-ниско ниво (CIL). Този език е стандартизиран от Microsoft. Това позволява да се пише на различни езици за една платформа. Mono допринася с това, че прави възможно изпълнението на нашите приложения върху най-широко използваните операционни системи и архитектури.

Статус на проекта

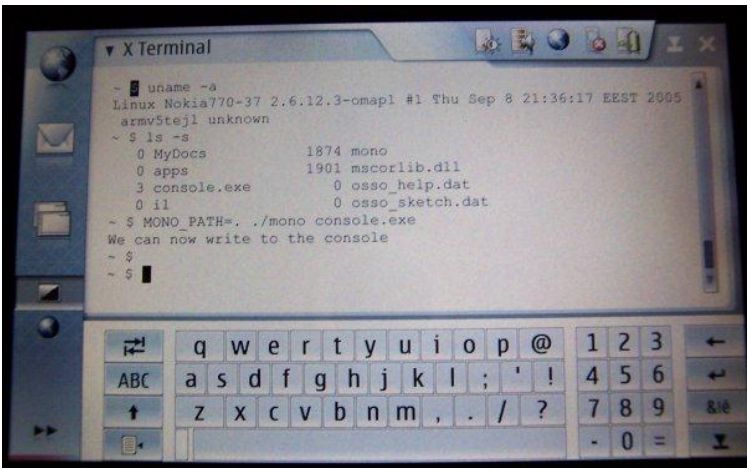
Текущата стабилна версия (ноември, 2006) е 1.1.13.8. Тя е от сериите 1.1.x, които се очаква да доведат до излизането на версията Mono 1.2, която ще включва някои компоненти, понастоящем не достатъчно стабилни за серията Mono 1.0. Версията 1.2 е планирана да излезе края на 2006 година и единственото, в което се различава от текущата версия е пълната поддръжка на Windows Forms. Ще бъдат включени и асемблита за от .NET Framework 2.0 за XML 2.0, ASP.NET 2.0, ADO.NET 2.0 и други. Има планове и за поддръжка и на .NET 3.0.

Официалният сайт на проекта предлага подробна и актуална информация относно статуса на Mono. Чрез системата за търсене по ключови думи, лесно можете да намерите всичко, от което се интересувате. Достъпни са множество ръководства за ползване на библиотеки свързани с Mono. Последните версии на проекта могат да бъдат свалени от адрес http://www.mono-project.com/Downloads.

Поддържани операционни системи и архитектури

Mono поддържа много операционни системи – Linux, Mac OS X, Sun Solaris, Free/Open BSD, Microsoft Windows. Компилира се както на 32-битови, така и на 64-битови архитектури. Mono е разработван предимно на Linux, затова Linux е най-добре поддържаната операционна система. Съществуват готови, компилирани пакети за дистрибуциите Suse, Red Hat 9.0, Fedora Core 3, Debian/GNU и Mac OS X. При BSD операционните системи Mono е включен в ports системата обикновено като **`lang/mono`**.

Поддържаните архитектури са x86, SPARC, s390, PowerPC, IA64. Mono работи и на устройства с ARM процесори:



Mono на Nokia-770

На картинката е показано конзолно Mono приложение, което изписва "Now you can write to the console", изпълнено върху Nokia-770, която е базирана на Linux.

Monoppix

Операционната система Monoppix (www.monoppix.com), базирана на Knoppix, включва в себе си Mono виртуална машина, компилатор и стандартни библиотеки, среда за разработка MonoDevelop, ASP.NET уеб сървър, библиотеката Gtk# за построяване на графични приложения, сървърът за работа с бази от данни MySQL и документация за Mono. Това е един лесен начин за изпробване на Mono, тъй като Monoppix се стартира само от CD и не изисква никакви допълнителни инсталации. Последната версия на Monoppix е свободна за изтегляне от http://www.monoppix.com/download.php.

Mono Live

Подобна на Monoppix дистрибуция е Mono Live. Тя е базирана на операционната система Ubuntu, която произхожда от Debian/GNU Linux. Сайтът на проекта е www.mono-live.com. Дискът, който можете да изтеглите от там, включва Mono, инструментът за разработка MonoDevelop и няколко пакета софтуер, работещ в средата на Mono.

Инсталиране и конфигуриране на Mono

Има два начина за инсталиране на Mono: чрез инсталация на готовите компилирани пакети или чрез компилиране на сорс кода на Mono.

Инсталиране на Mono върху Linux дистрибуции

Има няколко начина да се инсталира Mono под Linux.

Графичен инсталатор

Съществува графичен инсталатор за Linux, с който много лесно и бързо може да се инсталира Mono. Инсталаторът е достъпен от http://www.mono-project.com/Downloads и се стартира със следните команди:

```
% chmod +x mono-1.1.12.1_0-installer.bin
% ./mono-1.1.12.1_0-installer.bin
```

Графичният инсталатор се препоръчва за начинаещи потребители, както и за Windows/.NET програмисти, които искат да изпробват Mono под Linux с минимални усилия. Текущата версия включва и средата за разработка MonoDevelop.

Използване на готови дистрибутивни пакети

Mono се инсталира и чрез готовите компилирани пакети за различните Linux дистрибуции. За RPM базираните системи като Fedora и Suse Linux е препоръчително да се използват системите за инсталиране и обновяване **yum** или **yast2**. За Debian/GNU се използва **apt-get**. Чрез тези системи се избягват проблемите със зависимостите, защото освен самата програма се инсталира и всичко необходимо за нормалната ѝ работа.

Компилиране на сорс кода от дистрибутива на Mono

Ако компилирани пакети не са достъпни за дадена платформа, алтернативата е Mono да се инсталира чрез компилиране на сорс кода. За целта трябва да изтеглите архивите от Download страницата и да изпълните следните команди за разархивиране и компилиране на Mono:

```
# tar xzf mono-1.1.12.tar.gz
# cd mono-1.1.12
# ./configure
# make
# make install
```

Изтегляне на сорс кода от SVN хранилището

Указания относно изтеглянето на най-новия сорс код на Mono може да бъде намерен на страницата svn.myrealbox.com. Там се намира и SVN хранилището (repository) на Mono, което можете да разгледате с вашия уеб браузър.

SVN/Subversion е система с отворен код за контрол на версиите при съвместна работа в екип, подобна на CVS. SVN хранилището представлява нещо като файлов сървър за обмяна на файлове, с тази разлика, че то запомня всяка промяна на даден файл или директория. Това позволява проследяването на историята на даден файл, както и неговото възстановяване от по-стари версии при необходимост.

Ето примерни команди, с които можете да изтеглите Mono от неговото SVN хранилище (счита се, че имате локално инсталиран svn клиент):

```
# svn co svn://mono.myrealbox.com/source/trunk/mono
# svn co svn://mono.myrealbox.com/source/trunk/mcs
# svn co svn://mono.myrealbox.com/source/trunk/libgdiplus
```

Забележете, че краят на всеки ред е директорията, която ще бъде изтеглена. Може да посочите и други директории (или по-точно клонове).

Компилиране на сорс кода от SVN хранилището

Компилирането на изтегления сорс код от SVN хранилището става по следния начин:

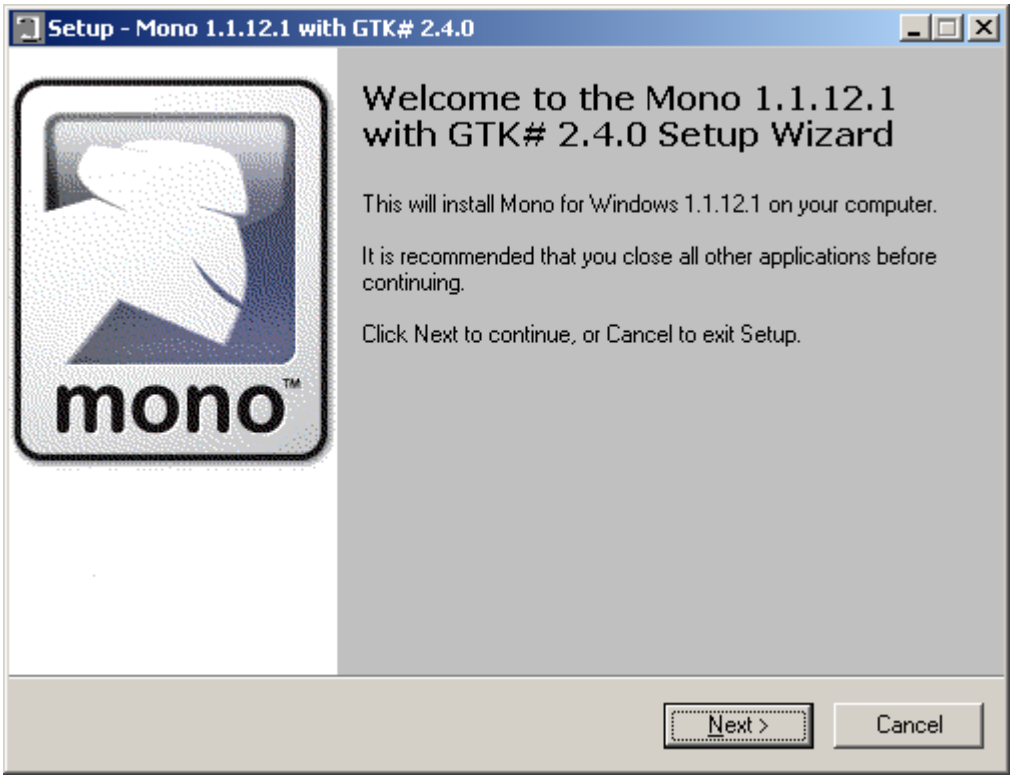
```
# cd mono
# ./autogen.sh --prefix=/usr/local
# make
```

Параметърът **--prefix** задава къде да се компилира кода (къде да се поставят файловете след инсталацията). Ако искате да обновите кода трябва само да влезете в съответната директория и да напишете:

```
# svn update
```

Инсталиране на Mono под Windows

За Windows проектът Mono предоставя инсталатор, обикновен Windows Setup, с включени Gtk# за разработване на графични приложения и сървъра XSP за ASP.NET.



При желание можете и сами да компилирате Mono, използвайки **cygwin**, Linux емулация за Windows, но за това се изискват по-задълбочени познания.

Инсталиране на Mono под Mac OS X

За MAC OS X има готов инсталатор. В него е включена платформата Cocoa#. По подразбиране Mono се инсталира се в **/Library/Frameworks**.

Инсталиране на Mono под FreeBSD

Инсталацията става изключително лесно, тъй като Mono е включен в ports системата на операционната система. Стартират се следните команди като потребител **root**:

```
# cd /usr/ports/lang/mono
# make install clean
```

Така Mono е инсталиран и готов за ползване.

BSD#

BSD# (http://www.mono-project.com/Mono:FreeBSD) е проект, който пренася Mono върху FreeBSD операционната система. Проектът работи върху поддръжката на съществуващите Mono ports във FreeBSD, върху пренасянето на нови Mono приложения, както и върху специфични проблеми свързани с интеграцията на Mono и FreeBSD.

Някои от ports, които се поддържат BSD#, не са включени в официалната FreeBSD ports колекция. По тази причина се налага обединение на двете колекции. Това става чрез използването на скрипт, поддържан от BSD#. Скрипта може да бъде свален от официалната страница на проекта и се стартира със следната команда:

```
# mono-merge
```

BSD# предоставя **lang/mono-svn** от ports колекцията си, чрез който може да бъде инсталирана текущата версия на Mono от SVN хранилището.

Препоръчително е при инсталиране да се ползва BSD#, защото съдържа винаги най-новата версия на Mono и на всички необходими инструменти и библиотеки.

Среди за разработка

Mono не предоставя директно среда за разработка. Затова разработчиците трябва сами да направят своя избор. Въпреки, че за Linux базирани операционни системи липсва такова мощно средство за разработка на .NET приложения като Microsoft Visual Studio за Windows, налице са няколко алтернативи.

MonoDevelop

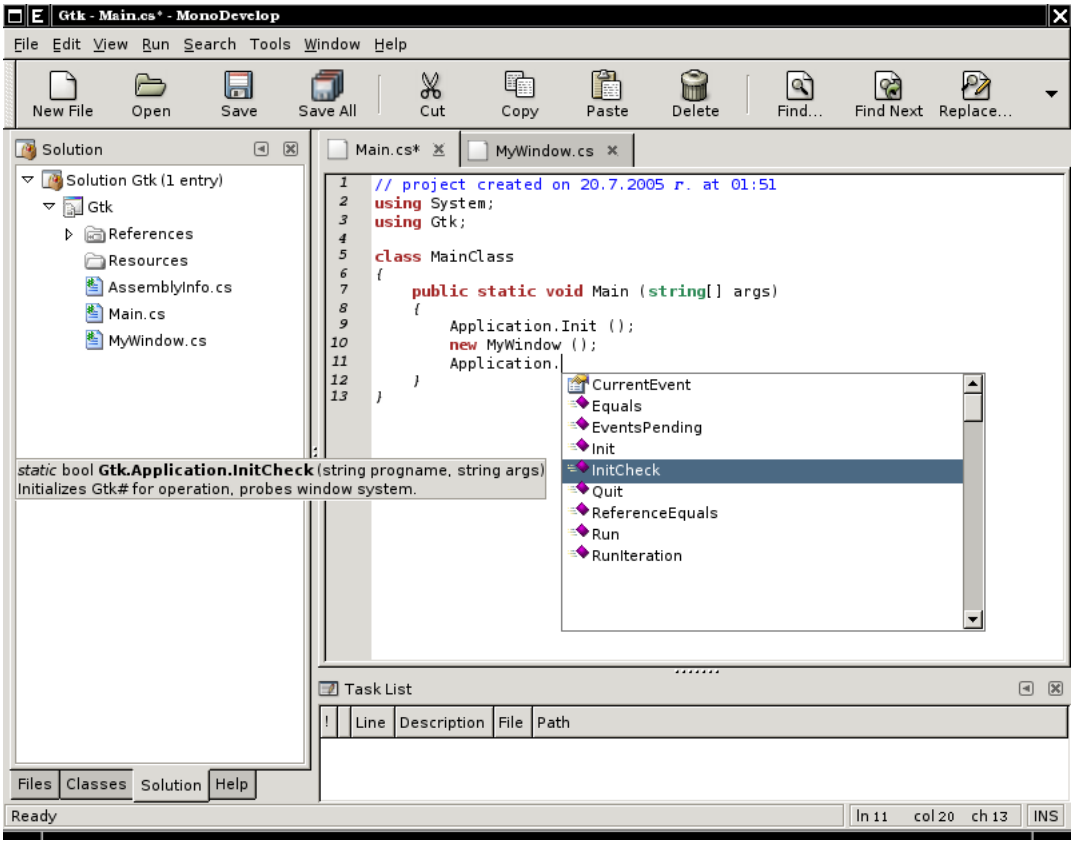
MonoDevelop (www.monodevelop.com) е среда за разработка с отворен код, която се разработва от екипа на Mono. Започната е като порт на SharpDevelop (среда за разработка на .NET приложения под Windows), а в момента се разработва като Gnome приложение. Настоящата версия е 0.9 (декември 2005). В последните си версии MonoDevelop поддържа плъгини – компоненти, добавящи функционалност, без да се налага прекомпилация.

MonoDevelop има вградена документация – Monodoc и дебъгер, автоматично допълване на код (code completion), подробен изглед на класовете, има и опция за конвертиране на Visual Studio проекти. Освен C# се поддържат и други езици като Java и Boo. Можете лесно да създавате

Gtk# и Gnome# проекти. Работи се по интеграцията на GUI дизайнер – в момента Glade3, а в бъдеще се очаква да бъде заместен от Stetic. На разположение е и Data Browser с който можете да прегледате съдържанието на базата данни с която работите. Дебъгерът за момента все още не е готов. За версия 1.0 е планирана поддръжката на **`gmcs`** и .NET 2.0, ASP.NET, както и системи за контрол на версиите (CVS и SVN).

При отварянето на VS.NET проекти може да се сблъскате с някои трудности. Вероятно ще трябва да подмените всички референции към асемблита, тъй като VS.NET ги записва като абсолютни пътища до Windows директориите. Това става лесно от **`Project Explorer -> References`** на MonoDevelop.

MonoDevelop се дистрибутира с някои от готовите инсталатори за Mono, както и с операционната система Monopix. Достъпен е и за FreeBSD чрез BSD# ports колекцията. Изискват се инсталирани GNOME библиотеки.



Eclipse

Eclispe е мощна и силно-разширяема среда за разработка, създадена от IBM. Основно се използва за разработване на приложения с Java, но поради гъвкавата и архитектура е възможно използването на много езици за програмиране с помощта на плъгини. За да използвате Mono и Eclispe

трябва да инсталирате плъгина "Improve C# Plugin for Eclipse". Информация за него може да бъде намерена на http://www.improve-technologies.com/alpha/esharp. Плъгинът поддържа създаване на C# файлове с основна структура, подчертаване на ключови думи, както и асистент за C# ключови думи. Предоставя се и възможност за компилация на C# файлове. От 2004 година няма нова версия на плъгина.

Emacs и Vim

Emacs и Vim са текстови редактори с общо предназначение, но могат да се използват и за сорс код редактори. С Emacs се използва C# editing mode, което може да бъде свалено от http://www.cybercom.net/~zbrad/DotNet/Emacs/. Поддържа оцветяване и подходящо подравняване. Vim поддържа също оцветяване на ключови думи.

X-Develop

Представлява комерсиален продукт, среда за разработка, написана на Java. Основно се използва за приложения, писани на Java, но поддържа и .NET.

KDevelop

Среда за разработка към проекта KDE, поддържаща много езици, включително и C#.

Какво включва Mono?

Mono предлага изградена инфраструктура, нужна за стартирането на .NET приложения. Mono включва компилатори, виртуална машина (CLR), съвместима със стандартите на ECMA (www.ecma-international.org) и множество библиотеки, както стандартните от Microsoft .NET Framework, така и допълнителни (Novell, Mono библиотеки и др.). Проектът предлага и браузър за документация (Monodoc).

Виртуална машина

Mono включва два инструмента за изпълнение на .NET асемблита: **`mono`** и **`mint`**. Те имплементират ECMA стандартите за Common Language Infrastructure (CLI) и включват Just-in-Time компилатор (JIT), Ahead-of-Time компилатор (AOT), компонент за зареждане на библиотеки (library/class loader), система за почистване на паметта (garbage collector), система за управление на нишките (threading system) и библиотеки за достъп до метаданни (metadata access libraries).

Интерпретаторът mint

Mint е интерпретатор за CIL байт код. Интерпретаторът изпълнява асемблита, съдържащи в себе си Common Intermediate Language код. Чрез следната команда се стартира .NET приложението **program.exe**:

```
% mint program.exe
```

Виртуалната машина mono

Виртуалната машина **mono** включва генератор на native код. Този генератор трансформира Common Intermediate Language в машиннозависим (native) код, което прави изпълнението на програмите бързо и ефективно. Генераторът работи в два режима Just-in-Time и Ahead-of-Time, като при режима Ahead-of-Time се прави предварителна компилация до native код, който се генерира еднократно и се използва винаги, когато се стартира съответното асемблi.

Виртуалната машина **mono** предлага опции, чрез които се настройва изпълнението на програми. Една от тези опции позволява редица оптимизации. Със следната команда се компилира входния файл до машиннозависим код с включени всички оптимизации.

```
% mono –O=all --aot program.exe
```

Повече информация относно използването на интерпретатора може да бъде намерена в помощната страница (man page) на Mono. Чрез следната команда се изписват на екрана кратко описание на възможните опции.

```
% man mono
```

Резултатът при изпълнение на една и съща програма с **mono** и с **mint** е един и същ. Разликата е в това, че **mint** прочита подадения му файл и го интерпретира в инструкции до процесора, докато при **mono** се използва JIT компилатора. След прочитане на инструкциите от входния файл **mono** извиква JIT компилатора, за да компилира тези инструкции до код на машинно ниво, след което този код се изпълнява. Забавянето при първо използване на **mono**, се дължи на това, че JIT компилаторът се нуждае от време, за да компилира съответната програма и да зареди генерирания машиннозависим код в паметта.

P/Invoke

P/Invoke, съкратено от "Platform Invocation Facility", позволява достъп до неуправляван код. Чрез този механизъм, интегрирането на C/C++ код във вашето Mono приложение става изключително лесно. Това дава възможност за обвиване на вече готови компилирани библиотеки писани на C/C++.

Компилатор за C# – mcs

Компилаторът на Mono - **mcs** е имплементация на ECMA-334 спецификацията за езика C#. Той е напълно съвместим с C# 1.0. В момента се работи и по втората версия на спецификацията на езика C# 2.0, но още не са имплементирани следните функционалности: extern директивата и поддръжка на приятелски асемблита (friend assemblies). Компилаторът на Mono за C# приема същите опции като компилатора на Microsoft за C#. Опциите могат да започват както с наклонена черта "/" , така и с тире "-". Всички специфични за Mono опции, които липсват при Microsoft компилатора, започват с "--". Компилацията се извършва от командния ред:

```
% mcs program.cs
```

Със следната команда се компилират рекурсивно всички C# файлове, започвайки от текущата директория:

```
% mcs -recurse:'*.cs'
```

Mono компилаторът е писан изцяло на C#. Така той може сам да компилира собствения си сорс код. Компилаторът може да бъде стартиран под Linux с Mono виртуалната машина, както и под Windows с .NET и Mono виртуалните машини.

Mono компилаторът за C# поддържа и някои оптимизации, като разгъване на константите (constant folding) и елиминация на неизползван код.

Както компилаторът, така и виртуалната машина на Mono имплементират ECMA стандартите, което ги прави напълно съвместими с Microsoft .NET Framework. Това позволява приложения, компилирани с **mcs** под Linux да се изпълняват под Windows и приложения компилирани с **csc.exe** да се изпълняват чрез виртуалната машина на Mono без да се налага прекомпилация.

За да постигнете такава съвместимост между Linux и Windows е нужно приложенията, които компилирате да не използват обръщения към платформено зависими функции чрез Win32 API или чрез P/Invoke.

Mono gmcs

Проектът Mono включва и компилатора **gmcs**, който ще замести **mcs** след версия 1.2 на Mono. В момента **gmcs** поддържа напълно последната ECMA спецификация (трето издание) и приложенията, разработени с него използват .NET 2.0 библиотеките. Поддържат се generics и другите нововъведения от .NET Framework 2.0.

Visual Basic .NET компилатор – mbas

MonoBASIC (**mbas**) е CIL компилатор за езика Visual Basic .NET. Базиран е на стара версия на **mcs** и в момента е бета версия. Приложения, писани под Windows и компилирани с Windows компилаторът за VB.NET, могат да се стартират с виртуалната машина на Mono, както и приложения компилирани с **mbas** могат да се стартират под Windows. Novell вече не поддържа разработката на **mbas**.

Mono асемблер и дизасемблер – ilasm и monodis

Проектът Mono включва и два инструмента за работа с Intermediate Language – асемблерът **ilasm** и дизасемблерът **monodis**. Асемблерът приема като входен параметър файл, съдържащ Common Intermediate Language и генерира файл (най-често с разширение **.exe** и **.dll**), който съдържа CIL байт код. Дизасемблерът на Mono генерира текстов файл с CIL инструкции от друг файл, съдържаш CIL байт код. Този файл може да бъде подаден на асемблера **ilasm** за генериране на асембли, което може да се стартира с виртуалната машина. Mono дизасемблерът се стартира със следната команда:

```
% monodis --output=program.il program.exe
```

Генерираният файл с име **program.il** можете да подадете на асемблера. Това става с командата:

```
% ilasm program.il
```

Mono дебъгерът – mdb

Mono предоставя и дебъгер за .NET приложения. Той може да бъде използван за дебъгване на управлявани и неуправлявани приложения. До момента дебъгерът е функционален, но е нестабилен и изисква тестване.

Изграден върху библиотека, притежаваща необходимите инструменти, Mono дебъгерът предлага два начина на ползване: чрез конзолната команда **mdb** и интеграция с MonoDevelop. На страницата http://deobald.glcn.com/wiki/index.php/Enabling_the_Debugger_Add-In можете да разберете как да използвате Mono дебъгера с MonoDevelop.

Mdb е Mono дебъгерът за работа от командния ред. По начина си на използване наподобява дебъгера за C/C++ приложения **gdb**.

За да се използва пълноценно **mdb**, програмата трябва да е компилирана със специална информация, улесняваща процеса на дебъгване. За да постигнете това, компилирайте вашата програма по следния начин:

```
% mcs -debug program.cs
```

Можете да стартирате **mdb**, за да дебъгвате вече компилираната програма, използвайки следната команда:

```
% mdb program.exe
```

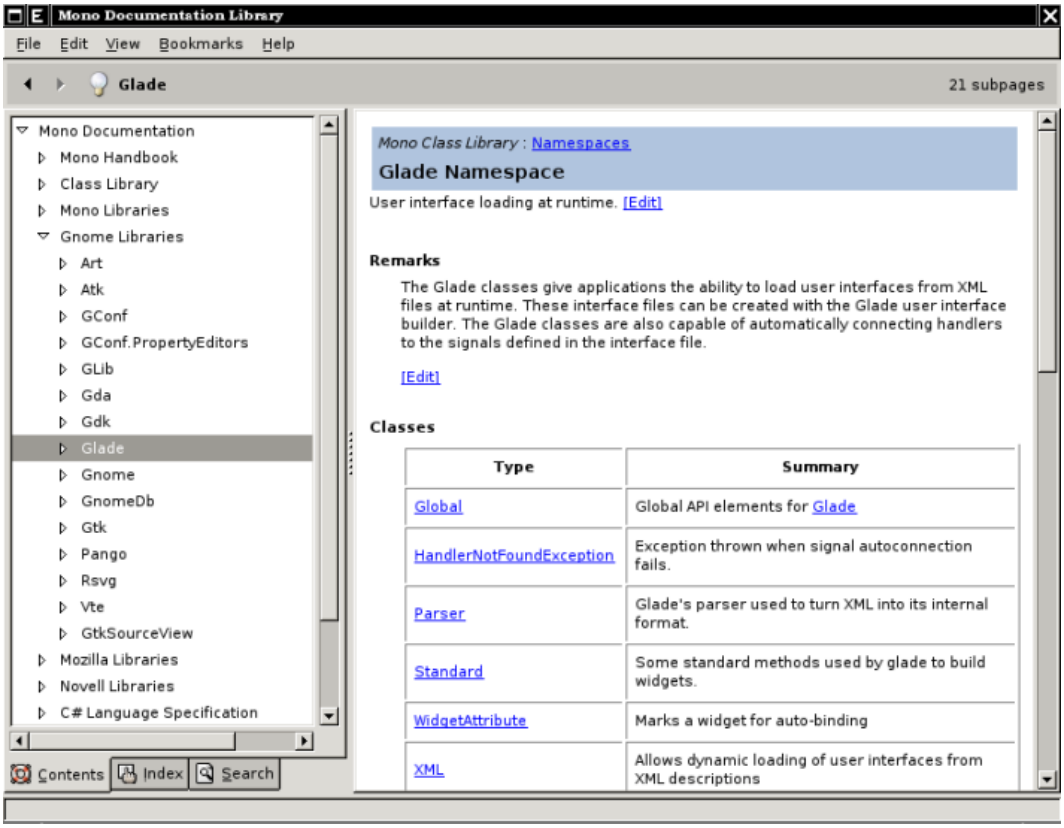
Чрез командите **continue**, **step**, **next**, **finish** можете да проследите изпълнението на вашата програма ред по ред.

Документацията Monodoc

Mono документацията може да бъде разгледана по три начина: на страницата http://www.go-mono.com/docs, чрез графичното приложение **monodoc** или чрез конзолното приложение **mod**.

Графичният браузър Monodoc

Monodoc е графичен браузър за документация на библиотеки. Съдържа описание на стандартните библиотеки, допълнителните Mono класове и класовете за графични приложения, съдържа спецификацията на езика C# и няколко ръководства. Позволява допълване на информацията.



При FreeBSD операционната система, браузърът Monodoc е включен в ports системата и се инсталира допълнително.

Monodoc се стартира със следната команда:

```
% monodoc
```

Mod

Съществува начин за преглед на документацията, без да се налага стартирането на графични приложения. За целта се използва конзолната команда **mod**. Информацията, която може да бъде получена от тази команда, е същата, както при използването на графичния браузър **monodoc**, с разликата, че е предоставена в текстов формат.

За да получите информация относно всички типове в пространството от имена **System**, изпълнете следната команда:

```
% mod N:System
```

Mono класовете

Освен основната библиотека с класове на Microsoft .NET Framework, Mono добавя и допълнителни пространства с класове. Ето по-важните от:

- В пространството **Mono.Security** се имплементират функционалности свързани със сигурността, като някои криптографски алгоритми, подписване на код и X.509 сертификати.
- **Mono.Cairo** е графична библиотека за връзка с библиотеката от ниско ниво за векторна графика cairo.
- **Mono.Math** добавя допълнителна функционалност за работа с математически операции, например генериране на прости числа.
- **Mono.Unix** предоставя интерфейс за работа с POSIX стандартизирани операционни системи. POSIX е набор от стандарти за писане на програми. Чрез тази библиотека, Mono осигурява набор от услуги, предоставени от POSIX стандартите като системни извиквания и сигнали. Пространството се използва и за локализация на конзолни и графични приложения. Следният пример показва как можем да вземем свободното дисково пространство под UNIX. Този код може да се изпълни само когато операционната система е UNIX подобна, как да направите това можете да разберете като погледнете в "Как да пишем преносим код?".

```
Mono.Unix.UnixDriveInfo info = new UnixDriveInfo("/");
Console.WriteLine("Disk Size: " + info.TotalSize);
Console.WriteLine("Free Space: " + info.AvailableFreeSpace);
```

- **Mono.Mozilla** библиотеките позволяват да се вмъкнат функционалности от браузъра Mozilla в Gtk приложенията ни.

- **`Mono.Gnome`** ни позволява да използваме възможностите на Gnome за създаване на потребителски интерфейси, работа с текст и различни виртуални файлови системи.
- Пространството **`Mono.Data`** предоставя връзка с множество бази данни като: PostgreSQL, MySQL, SQLite, Tds и Oracle.
- Библиотеката **`ICSharpCode.SharpZipLib`** предлага средства за работа с компресирани файлови и архиви.

В Mono, при имплементацията на стандартните библиотеки от .NET Framework понякога се добавят и допълнителни методи и свойства, разширяващи тяхната функционалност.

Полезни инструменти

Заедно с Mono, под Linux базирани операционни системи могат да се използват и някои изключително полезни инструменти, които са достъпни и под Windows върху Microsoft .NET Framework.

NAnt

NAnt (nant.sourceforge.net) е свободен инструмент за построяване за .NET приложения (build tool). Той улеснява изключително процедурата по компилация и изграждане на програмен пакет (package build) и предоставя редица допълнителни команди за връзка с операционната система, CVS и NUnit. Инструментът е аналог на популярния в Java средите Ant (ant.apache.org). Продуктът е базиран на .NET и работи и под Windows.

Информация относно инсталирането на NAnt за Mono е достъпна от страницата http://www.mono-project.com/NAnt_Installation.

NUnit

NUnit (www.nunit.org) е инструмент за създаване на unit тестове за всички .NET езици. Версия 2.2 на NUnit се разпространява заедно с Mono.

Графичното приложение Gnunit се използва за стартиране на NUnit тестове в графична среда.

Повече информация за инструментите, свързани с процеса на разработка на .NET приложения, може да се намери в темата "Помощни инструменти за .NET разработчици".

‘Hello Mono’ с Mono

Писането и изпълняването на приложения с Mono е толкова лесно, колкото и с Microsoft .NET Framework. Ще покажем стъпка по стъпка как се компилира и стартира примерна програма и как се използват дизасемблерът и Mono дебъгерът.

Сорс кодът

Създаваме файл с име **HelloMono.cs** със следното съдържание:

HelloMono.cs

```
using System;

class HelloMono
{
   static void Main()
   {
      string hello = "Hello Mono!";
      Console.WriteLine(hello);
   }
}
```

Компилиране

Така създаденият файл можем да компилираме чрез следната команда:

```
% mcs -debug HelloMono.cs
```

Файлът **HelloMono.cs** е вече компилиран. Резултатният файл е с име **HelloMono.exe** и съдържа CIL код. Допълнителната опция **-debug** генерира специална информация, която ще послужи при дебъгване.

Стартиране

Ще използваме виртуалната машина **mono**, за да изпълним компилираната програма. След успешно изпълнение на конзолата ще се изпише "Hello Mono!".

```
% mono HelloMono.exe
Hello Mono!
```

При първото изпълнение на програмата забелязваме леко забавяне. То е в резултат от работата на JIT компилатора, който компилира и зарежда native кода в паметта. При повторно извикване не се компилира втори път, поради което и изпълнението е малко по-бързо.

Дизасемблиране

Използваме дизасемблера на Mono **monodis**, за да разгледаме инструкциите към виртуалната машина, които компилаторът е генерирал.

```
% monodis --output=HelloMono.il HelloMono.exe
```

В резултат от горната команда получаваме файла **HelloMono.il**, който съдържа IL кода на дизасемблираното асембли:

HelloMono.il

```
.assembly extern mscorlib
{
  .ver 1:0:5000:0
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 ) // .z\V.4..
}
.assembly 'HelloMono'
{
  .hash algorithm 0x00008004
  .ver  0:0:0:0
}
.module HelloMono.exe // GUID = {3722559E-A214-4275-B057-
EF0E58CD4393}

.class private auto ansi beforefieldinit HelloMono
extends [mscorlib]System.Object
{
...
```

Дебъгване с mdb – Hello Mono ред по ред

За да илюстрираме работата с дебъгера **mdb**, ще използваме компилираната вече програмка **HelloMono.exe**. Започваме дебъгването със следната команда:

```
% mdb HelloMono.exe
Mono Debugger
```

Стартираме програмата с командата **run**:

```
(mdb) run
Starting program: HelloMono.exe
Process @4 stopped at #0: 0x003876be in HelloMono.Main()+0xe at
/home/ceco/Projects/dotnet/hello/HelloMono.cs:7.
   7            string hello = "Hello Mono!";
```

На последния ред се от изхода на дебъгера се изписва редът, който предстои да бъде изпълнен. С командата **next** пристъпваме към изпълнението на текущия ред:

```
(mdb) next
Process @4 stopped at #0: 0x003876d2 in HelloMono.Main()+0x22 at
/home/ceco/Projects/dotnet/hello/HelloMono.cs:8.
   8            Console.WriteLine(hello);
```

На последния ред отново виждаме текущата команда за изпълнение. След изпълнението на ред 7, с помощта на командата **print** можем да проследим състоянието на променливата **hello**:

```
(mdb) print hello
(System.String) "Hello Mono!"
```

Изписват се типът на променливата **hello** и нейната стойност.

Отново с командата **next** изпълняваме текущия ред от програмата и на екрана се отпечатва резултатът от изпълнението му. Продължаваме изпълнението на програмата ред по ред и излизаме от дебъгера с командата **quit**:

```
(mdb) next
Hello Mono!
Process @4 stopped at #0: 0x003876d9 in HelloMono.Main()+0x29 at
/home/ceco/Projects/dotnet/hello/HelloMono.cs:9.
   9    }
(mdb) next
Process @4 terminated normally.
(mdb) quit
%
```

ADO.NET и Mono

Mono има много ADO.NET Data Providers както за комерсиални сървъри за бази от данни (Microsoft SQL Server, Oracle, IBM DB2 Universal Database), така и за свободни такива (MySQL, PostgreSQL, SQLite). Някои от тях изискват допълнителни библиотеки, други са написани изцяло на C#.

Npgsql – Data Provider за PostgreSQL

Npgsql (http://gborg.postgresql.org/project/npgsql/projdisplay.php) позволява .NET приложения да се свързват и да обменят данни с PostgreSQL (www.postgresql.org) сървър за бази от данни.

PostgreSQL е една от най-добрите open-source бази данни и притежава много от възможностите на по-големите си комерсиални събратя. Всъщност PostgreSQL копира много от архитектурните и технологични подходи на Oracle (например езика PL/SQL), заради което понякога го наричат "клонинг на Oracle".

Npgsql Data Provider е имплементиран изцяло на C# и не изисква допълнителни библиотеки. С него можете да установите връзка с PostgreSQL сървър версия 7.x и по-нови, както и да създавате, променяте и изтривате данни. Класовете, нужни за работа с Npgsql, се намират в пространствата **System.Data** и **Npgsql**.

Ето пример за използването на Npgsql за връзка с PostgreSQL:

PostgreExample.cs

```
using System;
using System.Data;
using Npgsql;

public class PostgreExample
{
   public static void Main()
   {
      string connectionString =
         @"Database=mydb;
         Server=localhost;
         User ID=myusername;
         Password=mypassword";

      NpgsqlConnection dbcon =
         new NpgsqlConnection(connectionString);
      dbcon.Open();

      try
      {
         string sql = "SELECT name FROM Employee";
         NpgsqlCommand cmd = new NpgsqlCommand(sql, dbcon);

         // Reader example
         NpgsqlDataReader reader = cmd.ExecuteReader();
         using (reader)
         {
            while (reader.Read())
            {
               string name = (string) reader["name"];
               Console.WriteLine(name);
            }
         }

         // Data Adapter example
         NpgsqlDataAdapter adapter = new NpgsqlDataAdapter();
         adapter.SelectCommand = new NpgsqlCommand(sql, dbcon);
         DataSet resultDS = new DataSet();
         adapter.Fill(resultDS);
         DataTable tableEmployee = resultDS.Tables[0];
         foreach (DataRow row in tableEmployee.Rows)
         {
            Console.WriteLine(row["name"].ToString());
         }
      }
      catch (NpgsqlException sqlEx)
      {
        Console.Err.WriteLine(sqlEx);
```

```
        // Log the error ...
      }
      finally
      {
        // Clean up
        dbcon.Close();
        dbcon = null;
      }
   }
}
```

В примера се очаква в базата данни да има таблица **Empleyee** с колона **name** от тип символен низ (примерно **varchar(50)**). Ако искате да използвате успешно и кирилица, при създаването на базата данни трябва да зададете кодирането да бъде "**WIN**" или "**UNICODE**".

Примерът може да се компилира и стартира със следните команди:

```
% mcs TestExample.cs -r:System.Data.dll -r:Npgsql.dll
% mono PostgreExample.exe
```

MySQL Data Provider

В момента има два MySQL Data Providers: **ByteFX.Data.MySqlClient** и MySQL Connector/Net. Доставчикът на данни **ByteFX.Data.MySqlClient** не се разработва активно, но е включен в дистрибуциите на Mono. Препоръчва се използването на MySQL Connector/Net. Той се разработва и поддържа от MySQL AB (www.mysql.com), фирмата, която разработва сървъра MySQL, и трябва да бъде добавен допълнително към вашите библиотеки.

За да използвате MySQL Connector / Net Data Provider трябва да свалите съответната библиотека от страницата http://dev.mysql.com/downloads/connector/net/1.0.html.

Следващият пример показва начина на работа с MySQL Connector/Net:

MySQLExample.cs

```
using System;
using System.Data;
using MySql.Data.MySqlClient;

public class MySQLExample
{
  public static void Main()
  {
    string connectionString =
      "Data Source=MyServer;" +
      "Database=MyDatabase;" +
```

```
        "User ID=MyUser;" +
        "Password=MyPassword;";

      MySqlConnection dbcon =
        new MySqlConnection(connectionString);
      dbcon.Open();

      try
      {
        string sql = "INSERT INTO Test (id, text) " +
          "VALUES(1001, test)";
        MySqlCommand dbcmd = new MySqlCommand(sql, dbcon);
        dbcmd.ExecuteNonQuery();
      }
      catch (MySqlException sqlEx)
      {
        Console.WriteLine(sqlEx.Message);
        // Log the error ...
      }
      finally
      {
        // Clean up
        dbcon.Close();
        dbcon = null;
      }
    }
}
```

Компилираме и стартираме със следните команди:

```
% mcs MySQLExample.cs -r:System.Data.dll -r:MySql.Data.dll
% mono MySQLExample.exe
```

Повече информация относно MySQL Connector/Net е достъпна от страницата http://dev.mysql.com/downloads/connector/net/1.0.html.

OracleClient – The Oracle Data Provider

В пространството от имена **System.Data.OracleClient** се намира Data Provider за връзка със сървъра за бази от данни Oracle (www.oracle.com). До момента се поддържат Oracle 8i, 9i и 10g. За да използвате Oracle Provider се изисква библиотеката "Oracle OCI". **System.Data.OracleClient** може да бъде използван както под Linux, така и под Windows.

SqlClient – Data Provider за Microsoft SQL Server

ADO.NET Data Provider за работа с Microsoft SQL Server се намира в пространството **System.Data.SqlClient**. Поддържат се версиите Microsoft SQL Server 7, 2000 и 2005. Data Provider за Microsoft SQL Server е импле-

ментиран изцяло на C# и позволява и двата начина на автентикация: SQL Server Authentication и Integrated Authentication.

SqlClientExample.cs

```
using System;
using System.Data;
using System.Data.SqlClient;

public class SqlClientExample
{
   public static void Main()
   {
      string connectionString =
         "Server=MyServer;" +
         "Database=pubs;" +
         "User ID=MySqlServerUserId;" +
         "Password=MySqlServerPassword;";

      IDbConnection dbcon = new SqlConnection(connectionString);
      dbcon.Open();

      try
      {
         IDbCommand dbcmd = dbcon.CreateCommand();
         dbcmd.CommandText = "SELECT fname, lname FROM Employee";
         IDataReader reader = dbcmd.ExecuteReader();
         using (reader)
         {
            while(reader.Read())
            {
               string firstName = (string) reader["fname"];
               string lastName = (string) reader["lname"];
               Console.WriteLine("Name: {0} {1}", firstName,
                  lastName);
            }
         }
      }
      finally
      {
         // Clean up
         dbcon.Close();
         dbcon = null;
      }
   }
}
```

Примерът може да се компилира и се стартира със следните команди:

```
% mcs SqlClientExample.cs -r:System.Data.dll
```

```
% mono SqlClientExample.exe
```

Уеб технологиите в Mono

Нека сега разгледаме технологиите за изпълнение на ASP.NET уеб приложения и уеб услуги под Mono.

ASP.NET под Mono

Модулът mod_mono

Архитектурата на уеб сървъра Apache позволява използването на модули. Един модул отговаря за изпълнението на Perl скриптове, друг на PHP и т.н. В нашия случай модулът, обработващ ASP.NET заявките, се нарича **mod_mono**.

Инсталиране и конфигуриране на mod_mono

Най-добре е да намерите готов пакет за вашата дистрибуция. След като го инсталирате трябва да добавите във вашия Apache конфигурационен файл (**httpd.conf**) някои настройки, ако не са добавени автоматично. Има два начина за конфигурация на **mod_mono**.

Първи начин

От версия 1.1.10 на Mono в **mod_mono** е включена система за автоматична конфигурация, която позволява бързо и лесно управление на ASP.NET приложенията, изисква се минимална промяна на конфигурационния файл на Apache. Всичко, което е необходимо, е да включите модула и той автоматично ще обслужва ASP.NET приложенията, които се намират в уеб директорията на вашия уеб сървър. Това става като вземете файла **mod_mono.conf** от **xsp** или от **mod_mono** пакетите и посочите в **httpd.conf** пътя до него. Това става по следния начин:

httpd.conf

```
Include /etc/apache2/mod_mono.conf
```

В примера се предполага, че **mod_mono.conf** се намира в **/etc/apache2/** директорията. Много вероятно е тя да е различна на вашата система. Ако вече имате инсталиран модулът, тогава трябва да прибавите само:

httpd.conf

```
MonoAutoApplicationEnabled
```

Втори начин

Вторият начин е по-сложен. След инсталация конфигурационният файл на Apache (**httpd.conf**) трябва да бъде редактиран и в него да се добавят следните неща (ако не са били добавени автоматично):

httpd.conf

```
LoadModule mono_module modules/mod_mono.so
Alias /test "/usr/share/doc/xsp/test"
# Тук се дефинира свързване на виртуалната директория test с
# локалната /usr/share/doc/xsp/test. Тази команда е към Apache

MonoApplications "/test:/usr/share/doc/xsp/test"
# Тук задаваме същото като по-горе, но вече на mod_mono

<Location /test>
    SetHandler mono
</Location>
# SetHandler указва, че всички файлове от /test трябва да бъдат
# обслужени от mod_mono. Ако имате други файлове в същата
# директория, които трябва да бъдат обслужени от други модули,
# заменете SetHandler със следното:
# AddHandler mono .aspx .ascx .asax .ashx .config .cs .asmx .axd
```

/usr/share/doc/xsp/test е директорията, където **xsp** инсталира примерни **xsp** файлове, но може да бъде всяка друга в която има ASP.NET файлове.

Ако искате **mod_mono** да обслужва повече от един сайт, ще трябва да напишете нещо, подобно на следното:

```
LoadModule mono_module modules/mod_mono.so
Alias /test "/usr/share/doc/xsp/test"
Alias /personal "/home/user/mypages"
AddMonoApplications default (+ написаното на следващия ред)
"/test:/usr/share/doc/xsp/test, /personal:/home/user/mypages"
<Location /test>
    SetHandler mono
</Location>
<Location /personal>
    SetHandler mono
</Location>
```

Частта **AddMonoApplications** и директориите трябва да са на един ред. Обърнете внимание на запетайката в "**/test:/usr/share/doc/xsp/test, /personal:/home/user/mypages**", която разделя директориите на двата сайта. Освен това са необходими и двете настройки **<Location>**. Възможно е да има и няколко **mod_mono** сървъра.

Рестартиране на mod_mono

Съществува прост уеб интерфейс за рестартиране на **mod_mono**. За да го включите, можете да добавите следните настройки в **httpd.conf**:

```
<Location /mono>
  SetHandler mono-ctrl
  Order deny,allow
  Deny from all
  Allow from 127.0.0.1
</Location>
```

Рестартирането се налага, защото когато смените някой **dll** файл, трябва да рестартирате Mono процеса, който го обслужва. Друг начин за това е да рестартирате целия **httpd** демон (Apache):

```
% service httpd restart
```

При Debian/GNU Linux това става по следния начин:

```
% /etc/init.d/apache restart
```

Тестване на mod_mono

Сега остава да напишете в браузъра си http://localhost/test и да проверите дали се изпълняват примерните ASP.NET файлове или тези които вие сте указали.

Сървърът XSP

Вторият начин за хостинг на ASP.NET е чрез сървър, писан на C#. Ако използвате Linux, ще трябва да си компилирате и/или инсталирате **xsp** допълнително. Под Debian/GNU се инсталира пакетът **mono-xsp**, а под Fedora Linux – пакетът **xsp**.

Сървърът XSP се стартира по следния начин:

```
% mono /usr/local/bin/xsp.exe --root
/usr/local/share/doc/xsp/test/ --applications
/:/usr/local/share/doc/xsp/test/
```

- Опцията **--root** задава директорията, която ще стане настояща за **xsp**.
- Чрез **--applications** се задават две директории, разделени с две точки. Първата е виртуална уеб директория, а втората е съответстващата ѝ истинска директория.
- За повече информация погледнете man страницата на **xsp**.

Сега, в полето за адрес на браузъра, въвеждаме http://hostname:8080/index.aspx, където **hostname** е името на компютъра, който ползваме. Трябва да се заредят примерните **aspx** страници, които се разпространяват с Mono.

Xsp поддържа напълно ASP.NET 1.0 и ASP.NET 1.1. За ASP.NET 2.0 може да се използва Xsp2, който използва .NET 2.0 асемблитата. Поддържат се "главни страници" (master pages), клиентски обратни извиквания (client callbacks), както и много ASP.NET 2.0 контроли – менюта, дървета, гридове (gridviews) със сортиране на сървъра и при клиента.

Често срещани проблеми

Ако имате проблеми, следвайте следните стъпки, за да ги диагностицирате и отстраните:

- Проверете дали уеб сървърът има достъп до директориите, където се намират вашите файлове.
- Сложете вашето ASP.NET приложение, което искате да пробвате в **/usr/share/doc/xsp/test**, тъй като там правата за достъп са вече нагласени.
- Проверете **/var/log/apache2/error_log** за някакви проблеми със сървъра Apache.
- Проверете дали 8080, не е блокиран със защитна стена, или вече не се ползва от друго приложение (ако ползвате сървъра XSP).
- Значение имат големите и малките букви (например "Index" и "index" са различни идентификатори).
- Записвайте файла с вашия сорс код в UTF8 кодиране и използвайте след това опцията: **-codepage:utf8** на компилатора **msc**. В началото на ASP.NET файловете с разширение **.aspx** добавяйте следното:

```
<%@Page language="C#" compilerOptions="/codepage:utf8" %>
```

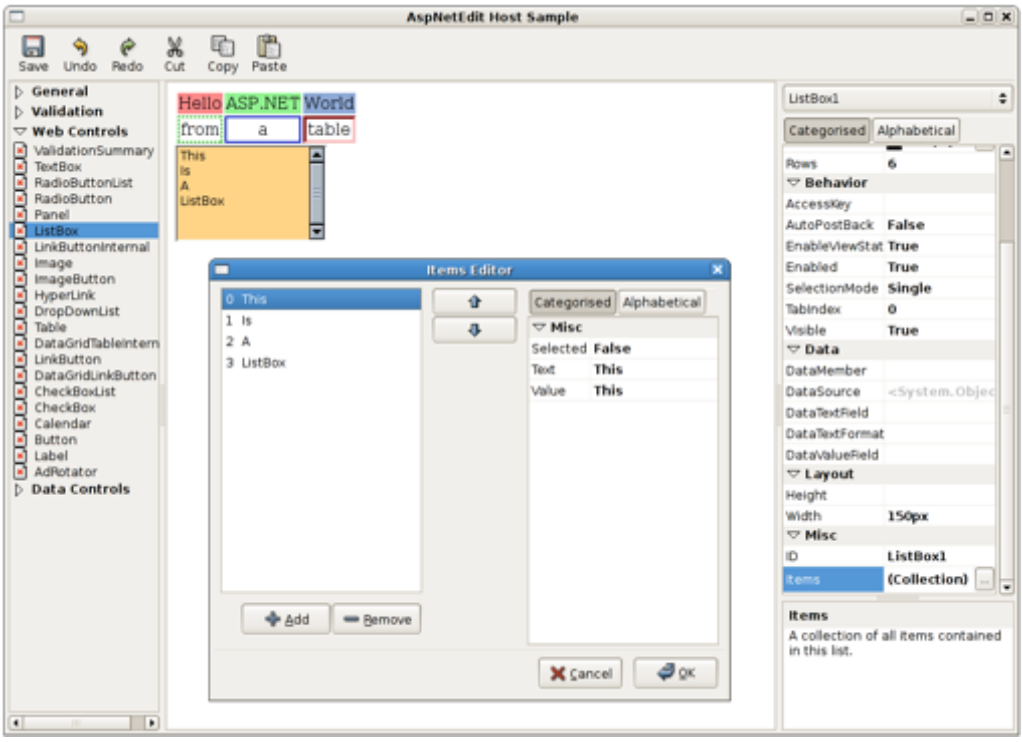
Прочетете "дежурните въпроси" (FAQ) за ASP.NET и Mono: http://mono-project.com/FAQ:_ASP.NET.

MonoDevelop и ASP.NET

Засега липсва вградена поддръжка на ASP.NET проекти в MonoDevelop, но се работи по въпроса. Планирано в MonoDevelop да бъде интегриран AspNetEdit – визуален дизайнер в ранна фаза на разработка, базиран на Mozilla редактора.

Засега няма компилирани версии на AspNetEdit и ще трябва да изтеглите кода и да го компилирате сами. Изтеглянето става със следната команда:

```
% svn co svn://mono.myrealbox.com/source/trunk/aspeditor
```



Уеб услуги

Уеб услугите са начин за отдалечено извикване на методи, подобно на RMI и RPC технологиите, но за разлика от тях, се използва платформено независими средства като SOAP протокола, правейки възможно приложения, писани на различни езици да си сътрудничат. Това позволява сложни и тежки изчисления да се правят на друга машина, облекчавайки тази, на която е стартирано приложението, използващо съответната уеб услуга. Mono предоставя всички инструменти необходими за лесното използване и създаване на уеб услуги.

Демонстрацията е направена на операционната система FreeBSD 5.4-STABLE, с помощта на XSP сървърът версия 1.1.10.0, Mono версия 1.1.10 и уеб браузърът Opera 8.51.

Създаване на уеб услуга

С настоящия пример ще покажем създаването, разгръщането (deployment) и тестването на примерна уеб услуга.

За целта създаваме файлът **MonoWebService.cs** със следното съдържание:

MonoWebService.cs

```
using System;
using System.Web;
```

```
using System.Web.Services;

[WebService (Description="Sample Web service with Mono")]
public class MonoWebService : System.Web.Services.WebService
{
   [WebMethod (Description="Says hello")]
   public string Hello(string name)
   {
      return "Hello " + name+ ", from Mono Web Service";
   }
}
```

Забелязваме познатите атрибути **[WebService]** и **[WebMethod]**, които определят уеб услугата и методите, които тя предоставя.

Компилираме със следната команда:

```
% mcs -r:System,System.Web,System.Web.Services \
MonoWebService.cs -t:library
```

Резултатът от компилацията е файлът **MonoWebService.dll**.

Създаваме директорията bin в текущата директория и преместваме в нея файла **MonoWebService.dll** със следните команди:

```
% mkdir bin
% mv MonoWebService.dll bin/
```

Сега се нуждаем от тестова страница за визуализиране на уеб услугата. Създаваме ASP.NET страницата с име **index.asmx** и със следното съдържание:

index.asmx

```
<%@ WebService Class="MonoWebService" %>
```

Остава ни да стартираме **xsp** сървърът и да разгледаме нашата уеб услуга:

```
% xsp --port 8080
xsp
Listening on port: 8080 (non-secure)
Listening on address: 0.0.0.0
Root directory: /usr/home/ceco/projects/dotnet/book/src/web
Hit Return to stop the server.
```

Въвеждаме http://localhost:8080/index.asmx в полето за адреси на нашия уеб браузър и след навигиране до тестовата форма можем да видим изхода от изпълнението на метода **Hello(…)**:

След изпълнението на метода, клиентът (в случая уеб браузърът) получава SOAP съобщение с дефинирани стойност и тип на върнатия резултат:

```
<?xml version="1.0" encoding="utf-8"?>
<string xmlns="http://tempuri.org/">Hello test, from Mono Web
Service</string>
```

Внедряване на уеб услуга

След като сме стартирали нашата уеб услуга, ще покажем как можем да я използваме в отделно приложение.

За целта, трябва да генерираме специален междинен клас (proxy), който ще се ползва от нашето приложение за връзка с уеб услугата. Това става с командата:

```
% wsdl http://localhost:8080/index.asmx -out:MonoProxy.cs
```

Необходимо е **xsp** сървърът да е стартиран по време на генерирането на междинния клас.

Ако разгледаме файлът **MonoProxy.cs** ще забележим два типа методи: синхронния **Hello(…)** и двата асинхронни **BeginHello(…)** и **EndHello(…)**. Разликата е, че при синхронното извикване на метод на уеб услугата, хода на програмата спира до завършването на метода, докато при

асинхронното извикване, резултата се получава в подадения обект от тип **System.AsyncCallback**.

Друг начин за генериране на междинен клас е след стартиране на **xsp** сървъра, въвеждаме адреса на уеб услугата в уеб браузъра и натискаме с мишката **Client proxy**.

След като имаме междинния клас (**MonoProxy.cs**) можем да използваме метода **Hello(…)** на уеб услугата. За целта създаваме файлът **MonoWebClient.cs** със следното съдържание:

MonoWebService.cs

```
using System;

class MonoWebClient
{
   static void Main()
   {
      MonoWebService ws = new MonoWebService();
      string hello = ws.Hello("test");
      Console.WriteLine(hello);
   }
}
```

Компилираме със следната команда:

```
% mcs -r:System.Web,System.Web.Services MonoProxy.cs \
MonoWebClient.cs -out:MonoWebClient.exe
```

След изпълнението на програмата, получаваме и очаквания резултат:

```
% mono MonoWebClient.exe
Hello test, from Mono Web Service
```

Ако разполагаме с WSDL (Web Service Description Language) файлът локално, можем да използваме инструмента **wsdl** по следния начин:

```
% wsdl MonoWebService.wsdl –out:MonoProxy.cs
```

Резултатът е междинният клас **MonoProxy.cs**, който можем да използваме при компилиране на приложения, използващи нашата уеб услуга. WSDL файлът може да бъде свален от съответния адрес на уеб услугата (чрез натискане с мишката на **Service Description -> Download** в уеб браузърът, виж screenshot-а).

Графични интерфейси в Mono

За разлика от Microsoft .NET Framework, Mono предоставя голям избор при разработката на графични приложения. С лекота можете да пишете приложения за KDE и Gnome, за Mac OS и Windows.

Всички демонстрации са направени на операционната система FreeBSD 5.4-STABLE. Версиите на инструментите са: Mono - 1.1.10, Glade Interface Builder - 2.12.1, Gtk# 2.4.0.

Windows Forms

Windows Forms е част от стандартната дистрибуция на Mono. Целта е да се имплементира напълно функционалността на пространството **`System.Windows.Forms`** в Microsoft .NET Framework.

Тъй като имплементацията на Microsoft използва платформено зависими извиквания за достъп до манипулатори на прозорци, шрифтове и т.н., се налага използването на Wine за стартиране на графични приложения използващи пространството от имена **`System.Windows.Forms`** (Wine (www.winehq.org) е емулатор, чрез който се стартират Windows приложения върху UNIX). Този подход, обаче, е неудачен, защото възникват редица проблеми, свързани с употребата на емулатора.

Поради описаната причина започва изграждането **`Windows.Forms`**, само от управляван код на базата на **`System.Drawing`**.

Ще илюстрираме как примерът, разгледан в темата за Windows Forms, се компилира и изпълнява под Mono. Нека си припомним как изглежда той:

MonoWindowsForms.cs

```
using System;
using System.Windows.Forms;

public class MonoWindowsForms: System.Windows.Forms.Form
{
   static void Main()
   {
      MonoWindowsForms sampleForm = new MonoWindowsForms();
      sampleForm.Text = "Mono Windows Forms";
      Button button = new Button();
      button.Text = "Close";
      button.Click +=
         new EventHandler(sampleForm.button_Click);
      sampleForm.Controls.Add(button);
      sampleForm.ShowDialog();
   }

   private void button_Click(object aSender, EventArgs aArgs)
   {
```

```
      Close();
   }
}
```

Компилираме и изпълняваме със следните команди:

```
% mcs MonoWindowsForms.cs -r:System.Windows.Forms.dll \
-r:System.Drawing.dll
% mono MonoWindowsForms.exe
```

На екрана ще се появи познатия прозорец и при натискане на бутона с текст "Close", прозореца се затваря.

Ако искате вашето графично приложение да се изпълнява на повече операционни системи, трябва да тествате дали работи под тях. Пълна (100%) поддръжка на **`Windows.Forms`** се очаква във версия Mono 1.2 в началото на 2006 г.

Gtk#

Gtk# (gtk-sharp.sourceforge.net) е един от често използваните инструменти за построяване на графични приложения с Mono. Библиотеката представлява .NET обвивка на библиотеката GTK+ и на някои Gnome библиотеки (gdk, atk, pango и т.н.). Приложения, писани с Gtk#, могат да работят върху Linux, Windows и Mac OS и много други.

Gtk# – пример

Нека да разгледаме следния пример за използване на Gtk#. Имаме просто C# приложение, базирано на Gtk#:

MonoGtkSharp.cs

```
using Gtk;
using System;

public class MonoGtkSharp
{
   static void Main()
   {
      Application.Init();

      Button button = new Button("Close");
      button.Clicked += new EventHandler(close);

      Window window = new Window("MonoGtk#");
      window.Add(button);
      window.ShowAll();

      Application.Run();
```

```
    }

    static void close(object aSender, EventArgs aArgs)
    {
        Application.Quit();
    }
}
```

Файлът се компилира и стартира с командите:

```
% mcs MonoGtkSharp.cs -pkg:gtk-sharp
% mono MonoGtkSharp.exe
```

На екрана се появява прозорец с единствен бутон с текст "Close", при натискането на който ще се прекрати изпълнението на програмата.

Програмен модел на Gtk#

Програмният модел на Gtk# е базиран на събития. Методът **`Application.Run()`** заставя приложението да чака до възникването на събитие. При възникване на определено събитие, се извиква функцията асоциирана с него.

В нашия пример асоциираме събитието, възникващо при натискане на бутона с текст "Close", с обработчик на събития, който извиква метода **`Close(…)`**. Това става чрез кода:

```
button.Clicked += new EventHandler(close);
```

Glade#

Glade# е библиотека, предоставяща свръзки (bindings) с библиотеката от ниско ниво libglade, и включва инструменти за зареждане на графични компоненти, създадени с Glade Interface Builder. Построените графични компоненти с Glade (glade.gnome.org) се запазват в XML формат, след което могат да бъдат заредени в сорс код и показани на екрана.

Инструментът Glade Interface Builder е включен в някои от готовите инсталации на Mono.

В следващия пример ще илюстрираме употребата на Glade#, Gtk# и Glade за построяване на просто графично приложение.

Използване на Glade

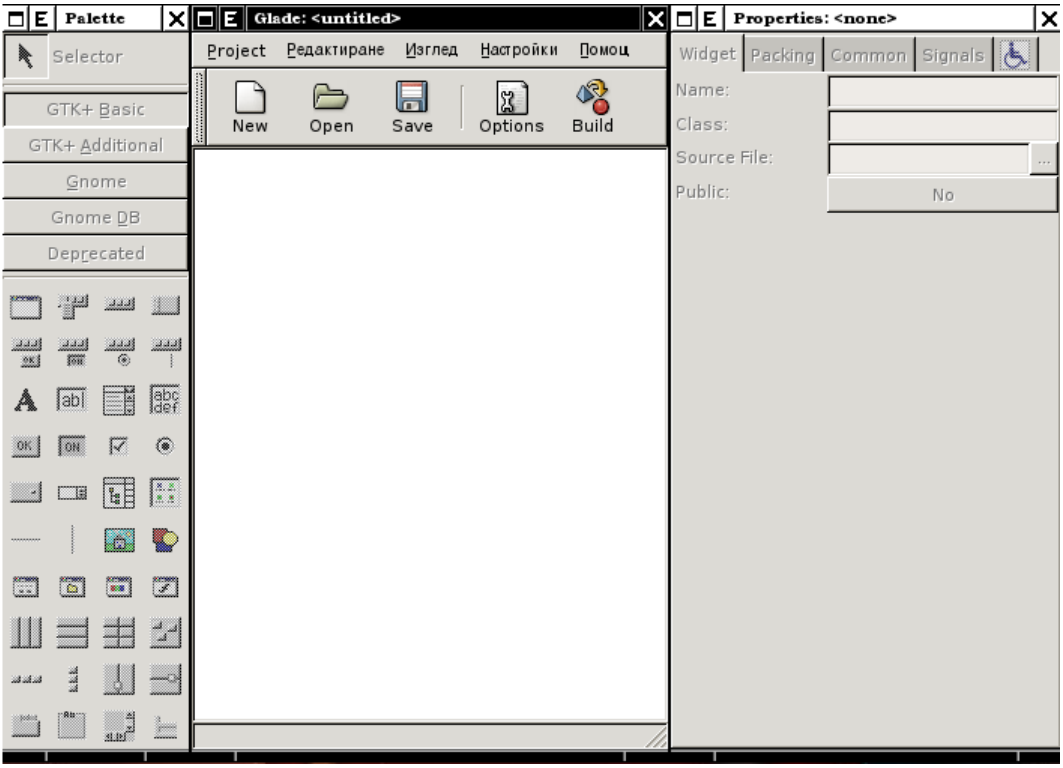
Нека разгледаме стъпките, които са необходими за създаване на Glade базирано приложение с Mono.

Стартиране

Стартираме Glade Interface Builder. Ако той е част от Mono инсталацията, трябва да се намира в неговата **bin** директория:

```
% glade-2
```

Ето как изглежда Glade Interface Builder:



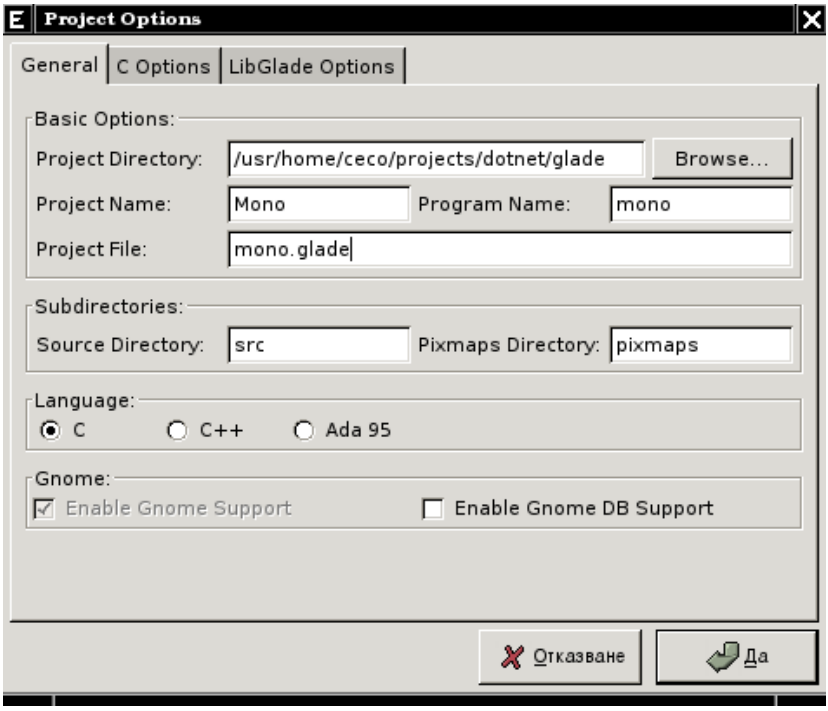
Той се състои от три прозореца:

- Основен - този прозорец служи за настройване на Glade проекти и е винаги видим.
- Палитра – съдържа всички визуални компоненти, които биха могли да се използват при построяването на графичния интерфейс.

- Аксесоари – съдържа информация относно текущия компонент (разположение, текст, сигнали, които ще се прихващат и т.н.).

Последните два прозореца са видими, само ако са селектирани менютата **`View | Show Palette`** и **`View | Show Property Editor`**.

Преди за започнем работа, създаваме нов GTK+ проект чрез **`New | New GTK+ Project`**:



Рисуване с палитрата

От палитрата натискаме бутона за създаване на прозорец.

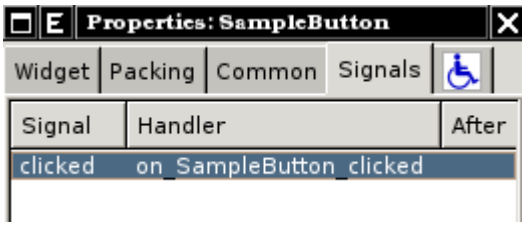
Нашият основен прозорец е създаден. Добавяме вертикална кутия с два реда чрез избиране на **`Vertical Box`** от палитрата и щракване с мишката върху създадения прозорец. В показания диалог въвеждаме цифрата 2

(създаваме два реда) и потвърждаваме с 'Да'. Добавяме по същия начин етикет (**Label**) в първия ред и бутон (**Button**) във втория ред на кутията. Прозорецът изглежда така:

Промяна на аксесоарите

Нека променим някои от аксесоарите на прозореца, бутона и етикета. Щракваме с мишката върху създадения бутон. Сега в редактора са заредени текущите аксесоари на бутона. Редактираме полетата **Name:** и **Label:** съответно със стойности "SampleButton" и "Press Me!".

Сега избираме менюто **Signals**. В полето **Signal:** ще посочим събитието, за което бихме искали да слушаме. Натискаме бутона означен с … (три точки) и избираме **clicked**. Натискаме бутонът **Add** и сигналът е добавен към списъка със сигнали, за които би могло да се слуша.



В колоната с име "Handler" е изписано името на функцията, асоциирана с обработчика на съответното събитие.

По аналогичен начин променяме аксесоарите на прозореца и етикета. Попълваме полетата **Name:** и **Label:** съответно със стойности "GladeWindow" и "Glade#" за прозореца и "SampleLabel" и "Before" за етикета.

Запазване на проекта

Запазваме проекта чрез менюто **Project | Запазване**, като избираме име на проекта "Mono" и произволна избрана директория.

Glade XML форматът

Нека разгледаме как Glade запазва построените графични компоненти. Следва част от файла **mono.glade**, намиращ се в директорията, където е създаден проекта:

mono.glade

```
<?xml version="1.0" standalone="no"?> <!--*- mode: xml -*-->
<!DOCTYPE glade-interface SYSTEM "http://glade.gnome.org/glade-
2.0.dtd">

<glade-interface>
<requires lib="gnome"/>
...
  <child>
    <widget class="GtkButton" id="SampleButton">
      <property name="visible">True</property>
      <property name="can_focus">True</property>
      <property name="label" translatable="yes">
        Press Me!</property>
      <property name="use_underline">True</property>
      <property name="relief">GTK_RELIEF_NORMAL</property>
      <property name="focus_on_click">True</property>
      <signal name="clicked" handler="on_SampleButton_clicked"
      last_modification_time="Wed, 20 Jul 2005 19:51:04 GMT"/>
    </widget>
    <packing>
      <property name="padding">0</property>
      <property name="expand">False</property>
      <property name="fill">False</property>
    </packing>
  </child>
...
```

Забелязва се йерархично изградената структура на компонентите. Всеки компонент се характеризира с аксесоари, описани в XML файл. Тези аксесоари могат да бъдат променяни чрез редакция на самия файл, както и чрез инструмента Glade Interface Builder. Във файла забелязваме и как се записва асоцииране на събитие с метод: **`<signal name="clicked" handler="on_SampleButton_clicked" last_modification_time="Wed, 20 Jul 2005 19:51:04 GMT"/>`**.

Как да покажем прозореца?

Ще покажем как да заредим XML файла чрез C# сорс код и как да визуализираме прозореца с Mono.

1. Създаваме файла **`GladeDemo.cs`** в директорията на Glade проекта със следното съдържание:

GladeDemo.cs

```
using Gtk;
using Glade;
using System;

class SampleGladeWindow
```

```
{
   private Glade.XML mGui;
   [Widget]
   private Gtk.Label SampleLabel;

   void on_SampleButton_clicked(object aSender, EventArgs aArgs)
   {
      SampleLabel.Text = "After";
   }

   public SampleGladeWindow()
   {
      mGui = new Glade.XML("./mono.glade", "GladeWindow", "");
      mGui.Autoconnect(this);
   }
}

public class GladeDemo
{
   static void Main()
   {
      Gtk.Application.Init();
      SampleGladeWindow window = new SampleGladeWindow();
      Gtk.Application.Run();
   }
}
```

2. Компилираме и стартираме с командите:

```
% mcs GladeDemo.cs -pkg:gtk-sharp -pkg:glade-sharp
% mono GladeDemo.exe
```

На екрана се показва прозорец със заглавие "Glade#". При натискането на бутона с надпис "Press Me!" текстът на етикета се променя от "Before" на "After".

Как работи примерът?

Примерният файл съдържа два класа: **GladeDemo** и **SampleGladeWindow**. Програмата започва изпълнението си от **Main()** метода на класа **GladeDemo**, където забелязваме стандартните за едно Gtk# приложение **Gtk.Application.Init()** и **Gtk.Application.Run()**.

Класът **SampleGladeWindow** представя прозореца GladeWindow, съставен с помощта на Glade Interface Builder. Член-променливата **mGui** се инициализира чрез **new Glade.XML("./mono.glade", "GladeWindow", "")**, където първият аргумент е абсолютният път на XML файла, който искаме да заредим. Вторият параметър е идентификаторът на компонентата, от която започва изграждането на частта от XML файла, която ще се покаже на екрана (в нашия случай зареждаме целия прозорец и всички компоненти в него). Чрез третия аргумент може да се задава област на превод при зареждане на XML файла (превеждат се заглавия и текст).

Чрез извикването на **mGui.Autoconnect(this)** свързваме всички събития със съответните им обработчици. Методът **Autoconnect(…)** свързва и полето **SampleLabel** със съответния компонент, дефиниран в XML файла (Забележете, че това поле е дефинирано с атрибута **[Widget]** и че името му е същото като на съответния идентификатор в XML файла).

Методът **on_SampleButton_clicked(…)** е асоцииран със сигнала **clicked**, който добавихме към бутона при работата с Glade Interface Builder.

Gnome#

Mono предоставя пространството **Gnome**, с помощта на което графичните приложения приемат облик (look-and-feel) на стандартни Gnome графични приложения. Пример за това е Gnome About диалогът, който стартиран с Mono изглежда така:

QT#

Qt# (qtcsharp.sourceforge.net) са класове за работа с библиотеката Qt, която е в основата на графичната среда KDE. Компилирането на Qt# базирана програма става по следния начин:

```
% mcs -r /complete_path/Qt.dll myprogram.cs
```

Тъй като Qt поддържа мобилни устройства (както и Mono), възможно е един ден да видим и Qt# приложения за тези устройства и да имаме нещо като еквивалент на Microsoft Compact Framework.

Cocoa# за Mac OS

Cocoa# (www.cocoasharp.org) е библиотека за изграждане на графични приложения за Mac OS X. Приложенията, изградени с Cocoa# са със специфичния облик на Mac OS X. Недостатък на тази библиотека е, че не може да се ползва за други операционни системи.

Как да пишем преносим код?

С Mono писането на платформено независим код е напълно възможно, стига да спазвате някои правила:

- Използвайте винаги релативни пътища, когато указвате път то файл или директория. Избягвайте платформено зависимите "**C:\Program Files**" и "**/usr/local/**".
- Mono не може да отваря файлове от вида \\server\mywork.txt. Съобразявайте се с това.
- Проверете дали не използвате obfuscated асемблита. Те прилагат специфични трикове за защита, които могат да объркат Mono и да доведат до неочаквани проблеми.
- Избягвайте използването на Windows регистрите (и класовете от пространството **Microsoft.Win32.Registry**).
- Ако използвате платформено зависими ресурси, винаги проверявайте дали съществуват. В случай, че не са достъпни, обработвайте адекватно възникналата грешка.
- При работа с текст за нов ред използвайте свойството **Environment.NewLine**, а не "**\r\n**" или "**\n**".
- Препоръчително е да компилирате в **Release** режим под Visual Studio .NET, тъй като VS.NET добавя много допълнителна **Debug** информация, която може да попречи на изпълнението на програмата под Mono.
- Записвайте файла с вашия сорс код в UTF8 кодиране и използвайте опцията: **-codepage:utf8** на **msc**. За ASP.NET страниците (файловете с разширение **aspx**) използвайте тага:

```
<%@Page language="C#" compilerOptions= "/codepage:utf8" %>
```

- Използвайте P/Invoke или други платформено зависими ресурси само в краен случай. Ако все пак няма алтернатива, използвайте някоя библиотека, достъпна за повече платформи. Следният код установява операционната система, на която се изпълнява приложението:

```
Type platformIdEnumType = typeof (PlatformID);
if (Enum.IsDefined(platformIdEnumType, "Unix"))
{
   if (Environment.OSVersion.Platform ==
      (PlatformID) Enum.Parse(platformIdEnumType, "Unix"))
   {
      Console.WriteLine("Platform: Mono on Unix");
   }
   else
   {
      Console.WriteLine("Platform: Mono on Win32");
   }
}
else
{
   Console.WriteLine("Platform: Microsoft .NET");
}
```

Можете да използвате примера, за да извършвате различни действия в зависимост от операционната система и CLR имплементацията, в която работи вашето приложение.

Програмиране на игри и Tao Framework

Когато става дума за графични среди, е редно да спомена и средствата за работа с графика в .NET и Mono. Ще се спрем на C#, DirectX, Mono, Tao, SDL.NET и Axiom.

Ако се занимавате с графика или имате влечение към разработването на игри, можете да правите това и с .NET, дори много по-лесно, отколкото с C++. Microsoft предоставят DirectX SDK, което може да бъде изтеглено свободно от сайта на Microsoft. То включва библиотека от .NET класове за работа с DirectX.

Работата с DirectX с C# е доста по–удобна от колко с C++. Добри примери за DirectX Managed Code може да намерите на www.codeproject.com.

Ако искате да пишете игра за .NET, най-вероятно бихте искали тя да работи върху всички .NET имплементации (Linux, MacOS X, BSD и Windows). Ако има изисквания за преносимост, най-добре е да изберете OpenGL пред DirectX. На следния адрес ще откриете различни проекти за разработка на .NET игри, преносими върху различни платформи: http://realmforgewiki.castlegobs.nl/index.php/Open-source_.NET_Game_Development_Collaboration.

Tao Framework

С Mono е свързан проектът Tao Framework (http://www.mono-project.com/Tao). Tao ви позволява да разработвате 2D и 3D графични приложения на C#. Засега Tao, не е стандартна част от Mono, но този въпрос е в

процес на обсъждане. Има компилирани пакети за Mono и Microsoft .NET Framework 1.1, за които има и инсталатор.

Ако сме запалили интересът ви, можете просто да изтеглите Tao, да го инсталирате и да пробвате примерите от **\ProgramFiles\Tao\Examples**, без да инсталирате нищо допълнително. Под Linux може да изтеглите официалната дистрибуция или да компилирате от SVN. Изтеглянето на кода става със следната команда под Linux:

```
% svn co svn://svn.myrealbox.com/source/trunk/tao
```

След компилация файлове търсете в **dist** папката. Може да се наложи за някои от примерите да инсталирате GLFW (glfw.sourceforge.net), което е свободен OpenGL Framework.

Примерите освен с изпълним файл, се доставят и като сорс код. Преди да започнете с примерите, запазете всичко, върху което работите. Възможно е компютърът ви да се рестартира или мониторът ви да угасне (в редки случаи).

Нека сега да стартираме и един от примерите. Под Linux с Mono трябва да изпълним командата:

```
% mono NateRobins.Starfield.exe
```

Под Windows просто отворете **C:\Program Files\Tao\examples** и щракнете два пъти върху файла **NateRobbins.Starfield.exe**.

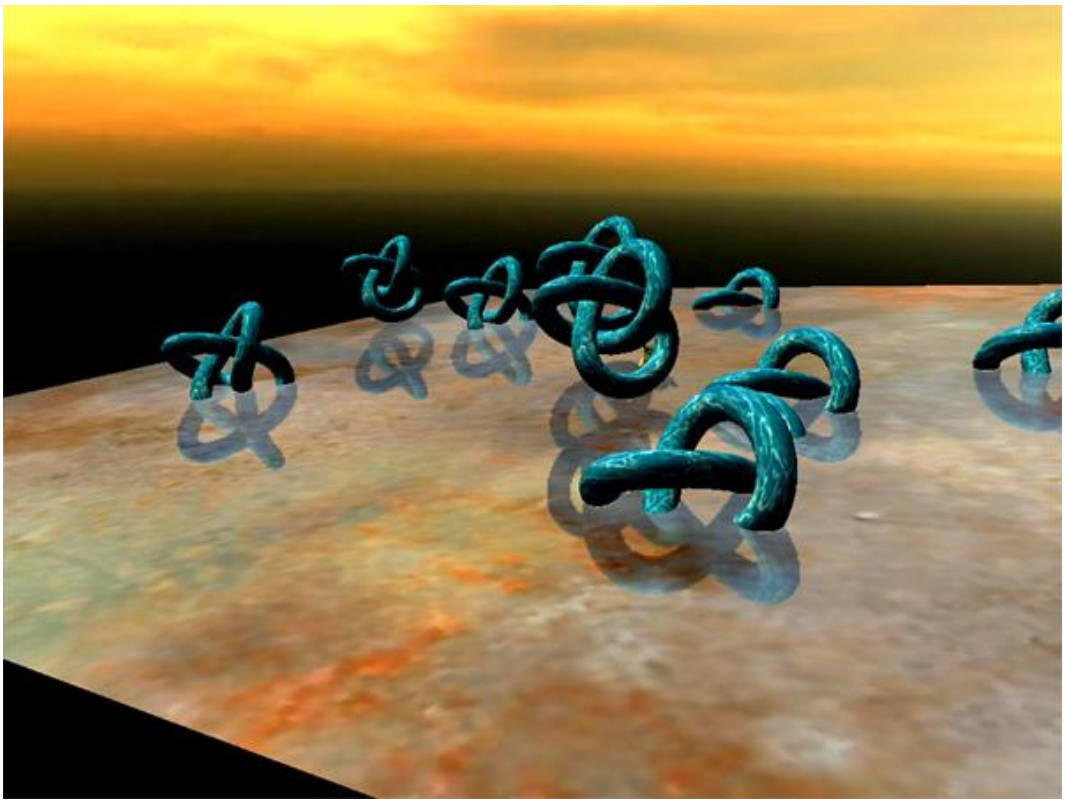
SDL.NET

SDL.NET (cs-sdl.sourceforge.net) е обектно-ориентирана .NET библиотека за разработка на игри, базирана на библиотеката SDL (Simple DirectMedia Layer – www.libsdl.org). Тя предоставя функционалност от високо ниво за работа с аудио, клавиатура, джойстик, шрифтове, различни графични формати, MPEG-1 филми, 3D OpenGL и други.

AXIOM

Има няколко инструмента (3D engines) за създаване на тримерна графика с C#. Един от водещите проекти в тази област е Axiom (www.axiom3d.org). Axiom е 3D инструмент (engine) с отворен код, базиран на Tao. Той е всъщност превод от C++ на C# на един от най-добрите графични енджини с отворен код – OGRE (www.ogre3d.org).

Ето как изглежда един сцена, визуализирана със средствата на Axiom:



Друг проект свързан с Axiom е RealmForge GDK (www.realmforge.com)

Java, Python, PHP и Mono

По идея .NET Framework интегрира разработката на много езици за програмиране в единна платформа с единен програмен модел и единни библиотеки за разработка. Mono отива дори по-далече – позволява интегриране на различни платформи и езици за програмиране в CLR.

Java за .NET CLR

Сайтът на проекта е www.ikvm.net. Идеята на проекта е разработчиците да могат да използват съвместно .NET и Java приложения, като ги изпълняват под Mono или Microsoft .NET Framework.

Това може да се постигне по два начина:

- Java класове да могат да се изпълняват от .NET виртуалната машина (CLR). Това се постига чрез конвертиране на Java bytecode към CIL в реално време.
- Java класовете се обгръщат от .NET асемблита (DLL файлове) и после те да бъдат реферирани от .NET проекти.

Python и PHP под Mono

Под Mono може да компилирате още Python (www.ironpython.com) сорс код и PHP скриптове (php4mono.sourceforge.net). За PHP има и още една .NET имплементация – www.php-compiler.net.

Упражнения

1. Инсталирайте Mono върху Linux, FreeBSD или Windows.
2. Опитайте да стартирате под Mono някое .NET асембли, писано и компилирано от вас преди това с Visual Studio .NET в Windows среда.
3. Опитайте да компилирате с Mono сорс кода от предходното асембли и след това да го изпълните.
4. Инсталирайте и стартирайте средата за разработка MonoDevelop. Опитайте да напишете с нея някое просто .NET приложение. Стартирайте и тествайте.
5. Реализирайте просто приложение, което извлича данни от MySQL база данни. Компилирайте и го изпълнете с Mono.
6. Инсталирайте и конфигурирайте **mod_mono**. Напишете просто ASP.NET уеб приложение с VS.NET и го стартирайте под Apache с **mod_mono**. Работи ли всичко нормално? Опитайте и с проста уеб услуга.
7. Стартирайте уеб приложението и уеб услугата от предходната задача под сървъра XSP.
8. Разгледайте документацията Monodoc. Намерете в нея помощна информация за разработка на Glade# приложения.
9. Опитайте да направите GUI приложение с Gtk#, Gnome# и Glade#.

Полезни Mono ресурси

1. www.mono-project.com – официалният сайт на проекта Mono.
2. www.gotmono.com – още един сайт, посветен на Mono.
3. www.gnomefiles.org – редица програми, писани за Mono и Gnome.
4. www.monodevelop.com – официален сайт на проекта MonoDevelop.
5. glade.gnome.org – сайтът на проекта Glade.
6. http://explore.openfmi.net/computers/programming/dotNET/Mono - страница, посветена на Mono, поддържана от Антон Андреев.

Използвана литература

1. Антон Андреев, Mono – свободна имплементация на .NET - http://www.nakov.com/dotnet/lectures/Lecture-25-Mono-v1.0.ppt

2. Brian Delahunty, Introduction to Mono – Your first Mono app – http://www.codeproject.com/cpnet/introtomono1.asp
3. Brian Delahunty, Introduction to Mono – ASP.NET with XSP and Apache – http://www.codeproject.com/cpnet/introtomono2.asp
4. The Mono Handbook - http://www.gotmono.com/docs

Глава 28. Помощни инструменти за .NET разработчици

Автори

Иван Митев

Христо Дешев

Необходими знания

- Базовите познания за .NET Framework
- Допълнителни познания, специфични за сферата на употреба на отделните инструменти

Съдържание

- Помощни инструменти за разработка
- Изследване на .NET асемблита с **.NET Reflector**
- Анализ на .NET асемблита с **FxCop**
- Генериране на код с **CodeSmith**
- Писане на unit тестове с **NUnit**
- Генериране на лог съобщения с **log4net**
- Работа с релационни бази от данни с **NHibernate**
- Автоматизиране на build процеса с **NAnt**
- Други помощни средства

В тази тема ...

В настоящата тема ще разгледаме редица инструменти, използвани в разработката на .NET приложения. С тяхна помощ можем значително да улесним изпълнението на някои често срещани програмистки задачи. Изброените инструменти ни помагат да разработваме по-качествени решения по-бързо, като могат значително да ни улеснят в писането на код и в поддръжката му. Всички средства, които ще разгледаме, са отлично допълнение към интегрираните среди за .NET разработка.

Помощни инструменти за разработка

Към този момент (май 2006 г.) съществуват стотици помощни инструменти, насочени към програмирането за .NET Framework. Те улесняват различни аспекти от разработката, подпомагайки решаването на често срещани типове проблеми. Голяма част от тези средства се използват с успех в реални проекти.

В тази тема няма как да опишем в детайли представители на всичките десетки категории, в които традиционно биват класифицирани такива помощни инструменти. За това ще се спрем само на най-важните области от разработката, като представим водещи, вече утвърдили се инструменти. Ще разгледаме само безплатни решения, по възможност проекти с отворен код.

За практиките и технологиите, които ще засегнем, са публикувани множество статии и книги. За голяма част от .NET инструментите, представители на съответните области от разработката, са написани десетки и дори стотици страници ръководства и статии. В тази глава ще обхванем само най-важните функции на разглежданите помощни средства. Ще изследваме сценариите за употребата им и по възможност ще дадем примери и код.

.NET Reflector

Reflector е браузър на .NET компоненти и декомпилатор. С негова помощ могат да бъдат разглеждани и претърсвани всички части на .NET асемблитата: техните метаданни, IL инструкциите, ресурсите и XML документацията. .NET Reflector (текущо версия 4.1.85.0) може да бъде изтеглен безплатно от http://www.aisto.com/roeder/dotnet – уебсайта на автора му Lutz Roeder.

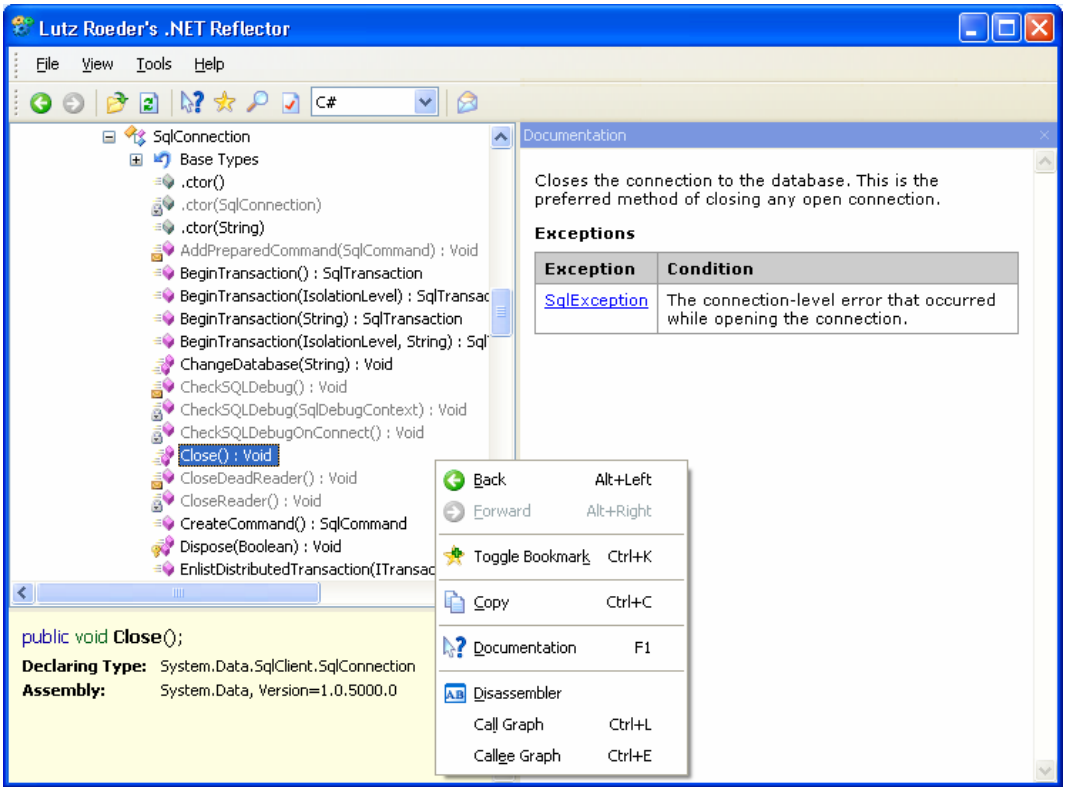
Функции

Основните функции, които Reflector предлага, са:

- Йерархичен изглед по асемблита и пространства от имена
- Търсене по име на типове
- Търсене по име на член-променливи и член-функции
- Преглед на XML документация
- Граф на извикванията
- Декомпилация в IL, C#, Visual Basic и Delphi
- Дърво на зависимостите
- Йерархия на базови и наследени типове
- Преглед на ресурси

- Бързо търсене в Google и в MSDN

Ето как изглежда главният екран на приложението:

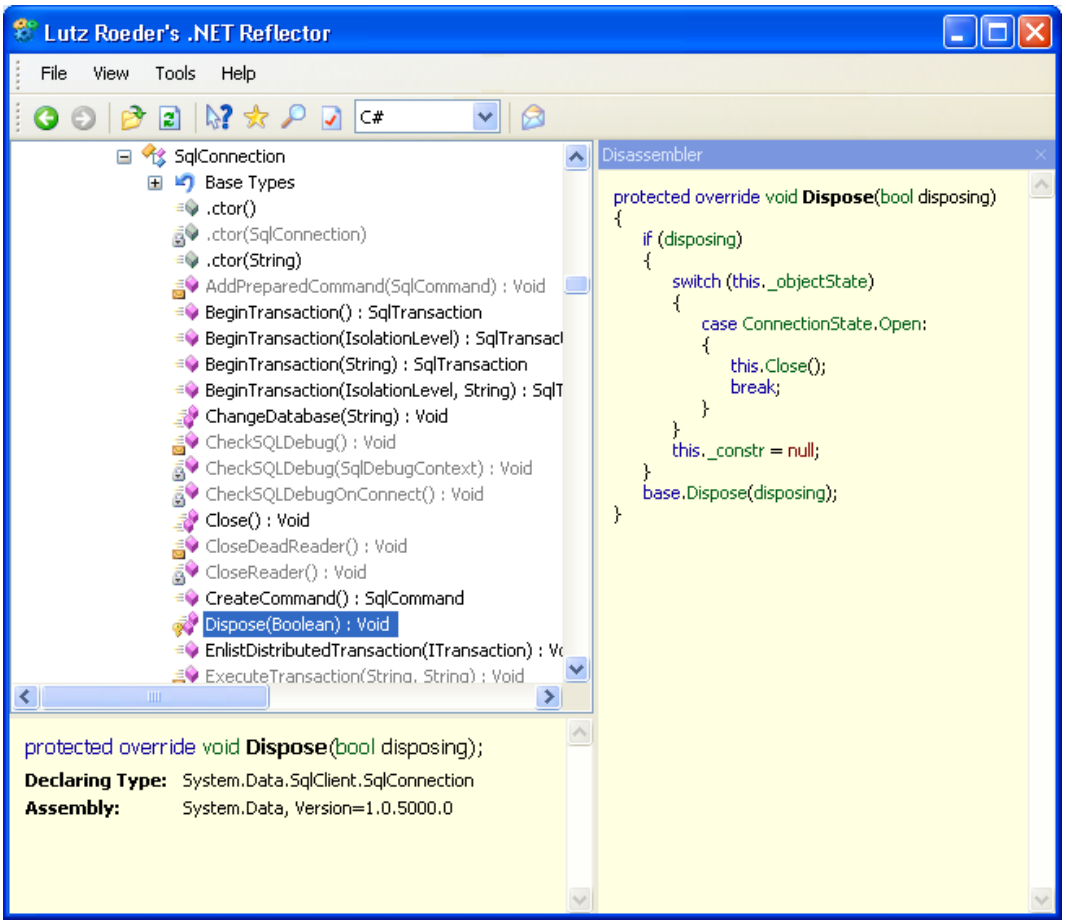


Навигация и търсене

Основното предназначение на Reflector е разглеждане на .NET асемблита и търсене в тях. По подразбиране при стартирането се зареждат основните асемблита на .NET Framework (можем да избираме между 1.0, 1.1, Compact Framework, …). Предлага се и меню за бърз избор на регистрираните в GAC асемблита. Reflector, разбира се, може да зарежда и произволни други асемблита, включително такива разработвани от нас.

Декомпилация на MSIL кода

Възможността за декомпилация е сред най-мощните и често използвани функции на Reflector. Тя е незаменим помощник там, където документацията на някое асембли е непълна, неясна или просто липсва. Ако искаме да разберем какво точно прави определен метод, трябва да позиционираме върху него и да стартираме Disassembler. Ще бъде генериран код на език по наше предпочитание: IL, C#, Visual Basic или Delphi. На фигурата по-долу е показан изход в C# за функцията Dispose() на System.Data.SqlClient. В прозореца с резултата реферираните функции и свойства се представят с хипервръзки. Така можем да се прехвърлим бързо в техните детайли от реализацията.



Граф на извикванията

Друга интересна функция на Reflector са графите на извикванията. Те действат в две посоки, т.е можем да видим както откъде се извиква даден метод, така и кои функции извиква самият той. Резултатите от анализа се визуализират във форма на граф, в който удобно могат да се проследят зависимостите.

Разширяемост

Reflector позволява лесно разширяване чрез механизма на добавките (add-ins). Проектът .NET Reflector Add-Ins, поддържан в GotDotNet (http://www.gotdotnet.com/workspaces/workspace.aspx?id=0f5846c3-c7aa-4879-8043-e0f4fc233ade) предоставя напътствия в насока създаването на добавки, демонстрирани чрез няколко примера с вариращa сложност.

Освен споменатия проект, редица разработчици независимо са създавали разширения за .NET Reflector. Подробен списък на добавки се поддържа на http://www.aisto.com/incoming/Reflector/AddIns/. Някои от по-любопитните включват:

- Reflector.CodeMetrics - анализиране на .NET асемблита и показване метрики за качеството на кода.
- Reflector.Graph - изчертаване на графи на зависимости за .NET асемблита и IL графи.
- Reflector.Diff – визуализиране на разлики между две версии на едно и също .NET асембли.
- Reflector.VisualStudio - вграждане на самия .NET Reflector във Visual Studio .NET 2003.
- Reflector.FileDisassembler - запис във файл на резултата от декомпилацията.
- Reflector.VSDisassembler - запис във файлове на резултата от декомпилацията и създаване на Visual Studio .NET 2003 проект.

FxCop

FxCop е безплатен инструмент, разработван от Microsoft, за статичен анализ на компилиран управляван код. Първоначално FxCop е бил създаден за вътрешните нужди на компанията. Той е допринесъл много за осигуряването на унифициран вид на .NET Framework API. Когато става ясно, че FxCop може да бъде полезен и в по-широка област от приложения, Microsoft отваря кода му. Текущата версия на FxCop е 1.32 и може да бъде изтеглена от http://www.gotdotnet.com/team/fxcop/.

FxCop анализира .NET асемблита и докладва за вероятни проблеми, свързани с множество аспекти на качеството на кода - проектирането, интернационализацията, производителността, сигурността. Голяма част от предложенията за подобрения, които FxCop дава, адресират нарушаването на някои от препоръките за програмиране и дизайн, публикувани в "**Напътствията при проектиране за .NET Framework от Microsoft®**" (http://msdn.microsoft.com/library/en-us/cpgenref/html/cpconnetframeworkdesignguidelines.asp). Този документ съдържа богат списък от насоки и правила на Microsoft за писане на надежден и лесен за поддържане код за .NET Framework.

FxCop е предназначен най-вече за разработчици на библиотеки от класове за .NET Framework, но голяма част от правилата му са приложими за различни типове приложения. Подпомагайки изучаването на утвърдени практики в .NET Framework, продуктът има и сериозна образователна стойност.

FxCop може да бъде интегриран в процеса на разработка на софтуер по няколко начина. За интерактивна работа той предлага приложение с графичен потребителски интерфейс (**`FxCop.exe`**). Инструментът за командния ред (**`FxCopCmd.exe`**) е подходящ за автоматизиране на build процеса и за интеграция с други инструменти.

Правила в FxCop

Правилата, идващи с FxCop, попадат в следните категории:

- Проектиране – откриване на вероятни недостатъци и проблеми при проектирането.
- Именуване – откриване на неправилно използване на малки и големи букви, колизии с ключови думи от различните езици и други въпроси, свързани с имената на типове, член променливи, параметри, пространства от имена и асемблита.
- Производителност – откриване на елементи от асемблитата, които водят до намалена производителност.
- Сигурност – откриване на програмни елементи, които правят асемблитата уязвими към злонамерени потребители / код.
- Употреба – откриване на вероятни недостатъци в асемблитата, свързани с начина на изпълнението на кода.
- Интернационализация – откриване на липсващи или неправилно използвани локализационни елементи в асемблитата.
- COM – откриване на проблеми, свързани с взаимодействието с COM обекти.

На всяко правило се определя ниво, показващо важността на открития проблем. Друга важна характеристика на правилото е степента на увереност, че коректно е определило ситуацията като проблем.

FxCop – графично приложение

В прозореца на FxCop се показват асемблитата и правилата, участващи в анализа, както и генерираните съобщения от проверката. По-долу на фигурата е показан вида на графичното приложение, чиито компоненти ще разгледаме.

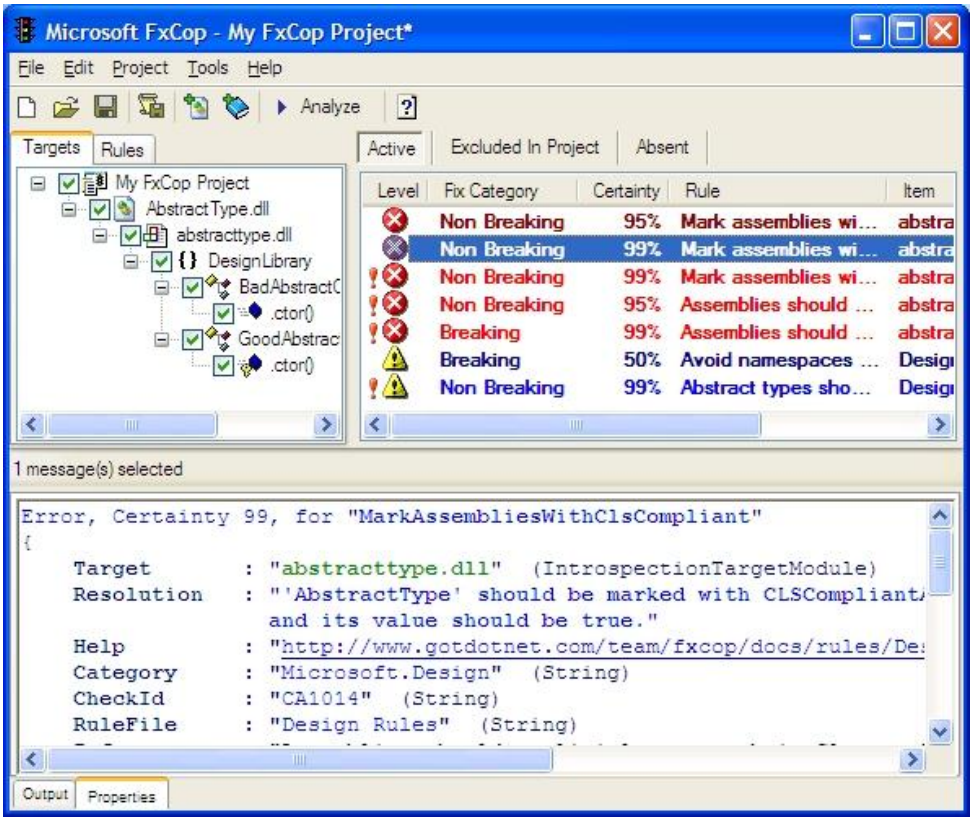
Компоненти на приложението

Работната площ е разделена на три основни региона: за конфигурацията, съобщенията и свойствата.

Регионът за конфигурацията, в лявата част на прозореца, показва в йерархичен изглед асемблитата и правилата. Тези два компонента дефинират FxCop проект. След като бъде конфигуриран, проектът може да бъда записан във файл с разширения (*.**FxCop**) и впоследствие зареден от приложението.

Регионът за съобщенията в дясната част на прозореца показва доклад със съобщенията, генерирани при анализа. Можем да филтрираме елементите по асемблита и по правила.

Прозорецът за свойствата показва в "Output" информация за предупреждения и грешки, а в "Properties" визуализира подробни данни за избрано асемблы, правило или съобщение.



Извършване на анализи с FxCop

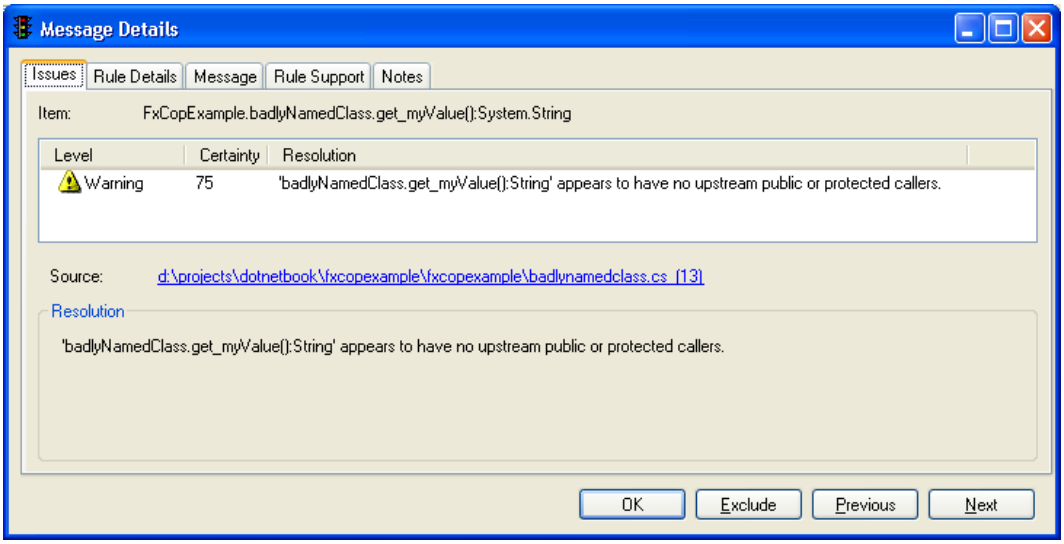
За да използваме FxCop, трябва да укажем едно или повече асемблита за анализ, както и едно или повече правила. По подразбиране в новия проект са избрани всички налични правила. След натискане на "Analyze" се създава и показва доклад със съобщения.

В региона на съобщенията можем да видим резултатите от анализа. От контекстното меню в региона за съобщенията, получаваме достъп до следните операции върху съобщенията:

- преглед на по-детайлни свойства
- копиране на данните в различни формати (**`.csv`**, **`.xml`**)
- изключване от бъдещи доклади

Извикването на прозорец с по-детайлна информация може да стане и чрез двойно щракване или с натискане на Enter. Визуализира се цялата налична информация за правилото и за проблемния участък от асемблито,

съпроводена с препоръчително действие. Ето как изглежда примерен прозорец с детайлите:



Изключване на съобщения

В определени случаи е възможно да ни се наложи да игнорираме някои съобщения. Може при проектирането на кода да сме направили съзнателен избор, който влиза в противоречие с някое правило. Възможно е и някое правило да е идентифицирало погрешно дадена ситуация като проблемна. За да не се появява в бъдещи доклади, излишното съобщение може да бъде изключено. При всяко изключване се запазват името на извършителя и незадължителен текст с причините.

Запазване на проект и доклад

Направените настройки по проекта могат да се запазят, така че да се ползват повторно във FxCop или да се импортират във FxCopCmd. Файловете за доклад, които по дефиниция съдържат набор от съобщения, могат също да се импортират във FxCopCmd или в друг FxCop проект. И двата формата са XML-базирани.

FxCopCmd – приложение за командния ред

FxCopCmd е подходящ за анализиране на асемблита в автоматизирана среда. Опциите за командния ред служат за указване на асемблитата, правилата и изходния файл. FxCopCmd не поддържа създаване и конфигуриране на проекти, нито изключване на съобщения. За тези операции, както и за по-детайлен контрол, се използва FxCop.

Ползи от употребата на FxCop

FxCop е един от инструментите, които ни помагат да създаваме по-добри приложения. Той идва с набор от утвърдени в Microsoft правила, но имаме

възможността да създаваме и добавяме собствени. Разработени са множество допълнителни правила, които лесно могат да бъдат намерени в интернет. Използването на FxCop ни дава повишена сигурност, че кодът ни се придържа към утвърдени практики за .NET разработката и практики, специфични за проекта ни.

С помощта на FxCop можем по-лесно да постигнем висока степен на последователност и унифицираност на кода, дори и в проекти с големи екипи. Автоматичното откриване на множество типични грешки и пропуски ще ни освободи повече време, което да използваме за по-важни въпроси от разработката. Да не забравяме, че FxCop не може да ни предпази от лошо проектиране и програмиране, както и не може да замести други полезни практики, като взаимния преглед на кода.

Използвана литература

- FxCop Documentation 1.312.0 - http://www.gotdotnet.com/team/fxcop/gotdotnetstyle.aspx?url=FxCop.html
- Anand Rao, Best Practices of Coding – http://www.c-sharpcorner.com/Code/2005/April/CodingPractices.asp

CodeSmith

CodeSmith (http://www.codesmithtools.com) е популярен генератор на код за всякакви програмни езици: C#, VB.NET, T-SQL и т.н. Той работи с шаблони, чиито синтаксис наподобява ASP.NET код. Текущата му версия е 3.1, а последната му напълно безплатна версия е 2.6. Тя може да бъде свалена от http://www.codesmithtools.com/download/codesmith_26.zip.

Предимството на комерсиалната версия е най-вече в наличието на средата CodeSmith Studio за бързо писане и тестване на шаблони, но тъй като те са текстови файлове, можем да ги създаваме и с помощта на обикновен текстов редактор.

Генериране на код

Нека преди да разгледаме възможностите на инструмента CodeSmith разгледаме генерирането на код като концепция: какво представлява, кога се ползва, с какво е полезно и т. н.

Какво представлява генерирането на код?

Генерирането на код представлява използване на програма за автоматично създаване на код, който после да бъде включен в сорс кода на друго приложение. Същата техника, освен за генериране на сорс код на програмни езици, има приложение и в други области, например в създаването на документация.

Генераторите на код получават като вход изисквания, които често се описват в XML нотация. Обработката на данните води до създаването на

един или няколко изходни файла. Генераторите с общо предназначение (какъвто е CodeSmith) най-често използват шаблони за описване логиката на работата, която извършват.

В практиката широко се използват и ръчно-създадени генератори. Такива помощни програмки се появяват, когато разработчиците предпочетат да ползват любимия си скриптов език за решаването на конкретния проблем, с който са се сблъскали. За прости задачи, този подход работи добре, но специализираните инструменти като CodeSmith имат своите предимствата. Те унифицират начина на представяне на логиката и така улесняват създаването, поддръжката и споделянето на шаблони.

Пасивни и активни генератори на код

Пасивните генератори служат за еднократно създаване на код, който впоследствие може свободно да бъде променян и настройван от програмиста, в случай че не отговаря напълно на изискванията на приложението. Този тип генератори дават "летящ старт" на разработката, но не можем да разчитаме на тях в по-нататъшния ход на проекта.

Активните генератори не само създават еднократно кода, а поемат отговорност да го поддържат. При всяка промяна на изискванията или входните данни се стартира пълно прегенериране на кода. Работата с активните генератори е добре да бъде напълно автоматизирана. Не се препоръчва употребата им в комбинация с ръчни модификации на изходния код, освен ако няма удобен и надежден механизъм, чрез който ръчните промени да бъдат запазвани при прегенерирането.

Предимства на генерирането на код

Генерирането на код ни носи много ползи. Нека разгледаме по-съществените от тях:

- **Продуктивност**: Генераторите на код могат да спестят часове и дни изпълняване на рутинни, повторяеми операции. В случай на променящи се изисквания към автоматично генерирана част от системата, с минимална промяна в шаблона и прегенериране, могат да се модифицират автоматично големи количества код.

- **Последователност**: Класовете, методите и променливите в изходния код са именувани унифицирано, което ги прави лесни за ползване. Логиката на реализацията на еднотипен код е последователна навсякъде в генерирания изход.

- **Елиминиране на дубликация**: Една от характеристиките на качествената реализация е намаляването на дубликацията на информацията в рамките на една система. Повторенията често водят до скъпо струващи модификации и затруднена поддръжка. Една идея или правило често неизбежно присъстват по няколко пъти в различни части от кода, базата от данни, документацията. Силно препоръчително е всяко знание в системата да има единствено, недвусмис-

лено, дефинитивно представяне. Следването на тази практика позволява промените и подобренията да бъдат извършени на едно място и автоматично да се разпространяват навсякъде. Генераторите на код понякога са най-практичното и дори единственото средство за справяне с такива проблеми. Чрез тях можем да създаваме и обновяваме автоматично части от кода и документацията на базата на представителните данни.

- **Абстракция**: Генераторите на код ни дават възможност да работим по-близо до предметната област, като създаваме абстракции, трудни за описване чрез език с общо предназначение като C# и после да генерираме от тях кода. Генераторите позволяват представяне на бизнес правилата и структурата на приложението във форма, удобна за преглед и анализ от хора, които не са програмисти. Добавянето на допълнителни нива на абстракция води до повишена гъвкавост и в други направления. Например само с промяна на шаблона за генериране, днес можем да произвеждаме C# код, а утре да преминем сравнително лесно към VB.NET реализация.

Проблеми с генерирането на код

Генерирането на код, както всяка друга техника, освен положителни страни има и някои недостатъци. Нека разгледаме някои от тях:

- За да се прилага ефективно трябва да се инвестират време и усилия за документиране, обучение и поддръжка.
- Генераторите могат да се окажат недостатъчно гъвкави за растящите нужди на проекта като да се усложнят до степен, при която поддръжката им става трудна.
- Генераторите могат да внесат известна нежелана сложност в процеса на разработка.
- Съществува опасност прекомерната им употреба да замести прилагането на солидно обектно-ориентирано моделиране.

Приложения на генераторите на код

Въпреки изброените недостатъци, генерирането на код остава мощна техника с широко приложение. Тя спечелва популярност първо при Java разработките. Към момента броят на генераторите на Java код е по-голям от този за всички останали езици, но .NET бързо наваксва, като се предлагат най-вече генератори на C# код. Използват се най-вече за достъп до бази от данни, за дефиниране на потребителски интерфейс и дори за създаване на цели GUI и уеб приложения. В .NET Framework 1.x са популярни и генераторите на силно типизирани колекции.

Въведение в шаблоните на CodeSmith

Синтаксисът на CodeSmith шаблоните много напомня на ASP.NET, където се използват разделители за отделяне на кода, изпълняван на сървъра, от

HTML и JavaScript кода, изпращан на браузъра. CodeSmith използва същата техника за отделяне на изпълнимия код на шаблона от кода, който ще бъде изведен като изход.

Използване на CodeSmith шаблони – пример

Ще разгледаме примерен шаблон за извеждане на имената на всички файлове в дадена директория, отговарящи на зададена файлова маска. В примера ще генерираме не програмен код, а обикновен текст.

FileSearchTemplate.cst

```
<%@ CodeTemplate Language="C#"  TargetLanguage="Text"
  Description="Simple template to show main syntax" %>
<%@ Property Name="Filter" Default="*.cst"
   Type="System.string" Category="Masks"
   Description="Mask for files in the directory" %>
<%@ Assembly Name="SchemaExplorer" %>
<%@ Assembly Name="System.Design" %>
<%@ Import Namespace="SchemaExplorer" %>
<%@ Import Namespace="System.IO" %>
FileSearchTemplate used to show syntax and
structure of template.
<%= DateTime.Now.ToLongDateString() %>

<%
// Comments within code delimiters or script blocks
// are made using the Language syntax (e.g. C#)
Response.WriteLine
   ("List of files in template directory (using mask "
      + Filter + ")");
DisplayDirectoryContents(Filter);
Response.WriteLine(">> Code Generation Complete.");
%>

<%-- Codesmith style comment --%>

<script runat="template">
// Iterates through the current directory and displays
// a list of the files that conform to the supplied
// mask.
public void DisplayDirectoryContents(string sFilter)
{
    string[] dirFiles = Directory.GetFiles
       (this.CodeTemplateInfo.DirectoryName, sFilter);

    for (int i = 0; i < dirFiles.Length; i++)
    {
         Response.WriteLine(dirFiles[i]);
     }
}
```

```
</script>
```

Изходът на **FileSearchTemplate.cst** е следният:

```
FileSearchTemplate used to show syntax and
structure of template.  18 July 2005
List of files in template directory (using mask *.cst)
C:\Program Files\Codesmith\Samples\FileSearchTemplate.cst
C:\Program Files\Codesmith\Samples\StoredProcDB.cst
C:\Program Files\Codesmith\Samples\StoredProcs101.cst
>> Code Generation Complete.
```

Как работи примерът?

Изходният текст съдържа заглавието на шаблона, датата на генерирането и имената на файловете, открити по указана маска. Списъкът с изброените файлове получаваме като резултат от извикване на метода **DisplayDirectoryContents(…)**, който сме реализирали на езика C#. Низът, генериран от **DisplayDirectoryContents(…)** се добавя в изходния текст чрез метода **Response.WriteLine(…)**.

За шаблоните свойствата са това, което са входните параметри за функциите. **FileSearchTemplate.cst** дефинира единствено свойство **Filter**, в което се указва маска за файла (примерно ***.doc**). По подразбиране сме заложили стойността ***.cst** (***.cst** е разширението за CodeSmith шаблоните).

В примера видяхме как в шаблоните могат да се ползват възможностите на произволен .NET клас, в случая **System.IO.Directory**. Също така демонстрирахме как може да се извлече информация за текущия шаблон (обект от тип **CodeTemplate**) чрез използване на свойството **this.CodeTemplateInfo**. Класът **CodeTemplateInfo** предоставя свойства за **DateCreated**, **DateModified**, **Description**, **DirectoryName**, **FileName**, **FullPath**, **Language** и **TargetLanguage**.

Директиви

Елементите **<%@ %>** се използват за указване на свойства и директиви на ниво шаблон.

- **CodeTemplate** директивите се използват за указване на скриптов език (в примера това е C#), целевия език и описанието.
- Чрез директивата **Property** се дефинират свойствата на шаблона. Те могат да бъдат реферирани в скрипта, като се използва името им (атрибута **Name**). Може да им се задават и тип, стойност по подразбиране, описание и категория.
- Директивата **Assembly** позволява реферирането на външни .NET асемблита.

- Директивата **`Import`** е еквивалентна на командите: **`using`** в C# и **`Imports`** във VB.NET.

Елементи за код

Използването на елементи за код е подобно на употребата им в ASP.NET:

- **`<% %>`** се използва за код, който няма да се появява директно в изхода от шаблона
- **`<%= %>`** се използва за код, който връща стойност за шаблона. Тази стойност трябва да е от текстов тип.
- **`<script runat="Template"> </script>`** е за включване на методи, използвани от шаблона (като **`DisplayDirectoryContents(…)`** от примера). Така се намалява количество код в <% %> елементите и се подобрява четивността на кода.

Коментари

- Коментарите, намиращи се в елементите за код, използват синтаксиса на езика, указан в директивата **`CodeTemplate`** (т.е за C# бихте използвали //, а за VB.NET апострофи ').
- За коментари, които са вътрешни за шаблона и не са предназначени за изходния код се използват **`<%-- --%>`** елементи.

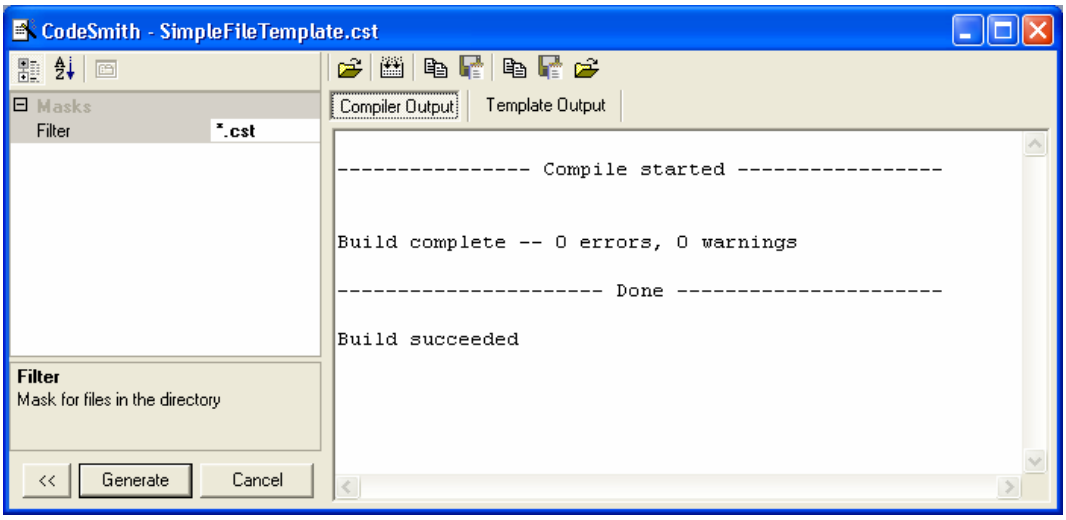
CodeSmith приложения

Да разгледаме двете основни приложения, които се доставят с инструмента CodeSmith – графичният потребителския интерфейс и конзолният вариант.

CodeSmith Explorer

CodeSmith Explorer (**`CodeSmith.exe`**) е приложението, с което можем интерактивно да стартираме генерирането на код от избран шаблон. Ако бъде стартирано без параметри, то показва наличните шаблони, организирани или по директории или по целеви език. След като изберем желания шаблон (можем и да го зададем директно като команден аргумент при стартиране на приложението) се появява прозореца от фигурата по-долу.

Преди да пуснем шаблона да се изпълнява, можем да го компилираме, за да проверим синтактичната му коректност. В лявата част на прозореца са параметрите на шаблона. Указването на стойностите им може да се извърши или на ръка или чрез зареждане от XML файл. След натискане на "Generate" шаблонът се прилага върху указаните параметри и в "Template Output" се появява резултата. Можем да копираме изходния код или да го запазим като файл.



CodeSmith Console

Конзолното приложение **CodeSmithConsole.exe** е удобно за автоматизация – примерно включване на генерирането на код като част от build процеса. Може да му подадем път до шаблон, път до XML файл със свойства и да укажем къде да се запише резултатът. Ето как изглежда един примерен XML файл със свойства:

FileSearchTemplateParams.xml

```
<?xml version="1.0" encoding="us-ascii"?>
<codeSmith>
  <propertySet>
    <property name="Filter">*.doc</property>
  </propertySet>
</codeSmith>
```

Извикването на приложението може да стане със следната команда:

```
C:\Program Files\CodeSmith\v2.6>CodeSmithConsole
/template: Samples\FileSearchTemplate.cst
/properties: FileSearchTemplateParams.xml
/out: result.txt
```

Друга важна характеристика на конзолното приложение е възможността за вмъкване на резултата от генерацията в указан регион на изходния файл. Нужно е само да добавим аргумента **/merge:regionName** и да осигурим, че в изходния файл присъства секцията от вида **#region regionName #endregion**. По този начин става възможно съжителстването в рамките на един файл на автоматично генериран код и такъв писан на ръка.

Използвана литература

- Code Generation: The One Page Guide - http://www.codegeneration.net/files/JavaOne_OnePageGuide_v1.pdf
- Dave Thomas Interview on Code Generation - http://www.codegeneration.net/tiki-read_article.php?articleId=9
- Code Generation with CodeSmith, Brian Boyce - http://msdn.microsoft.com/vstudio/default.aspx?pull=/library/en-us/dnhcvs04/html/vs04e5.asp
- CodeSmith Tutorial - http://www.codesmithtools.com/features/tutorial.aspx

NUnit

NUnit е среда за писане и изпълнение на unit тестове за .NET. Тя има два компонента: библиотека, която използваме при писането на тестовете и инструменти за изпълнението им. Такива инструменти, известни със събирателното име xUnit, има пренесени за различни среди и езици за програмиране. Всички те имат подобен дизайн, имитиращ този на първообразите: sUnit за Smalltalk и JUnit за Java. Първата версия на NUnit е почти директно пренесена от Java варианта, докато втората използва идиоматичните за .NET атрибути при маркирането на тестовете. Въпреки че съществуват други библиотеки за писане на unit тестове за C# и .NET изобщо, NUnit е най-разпространена. Тя се е превърнала де факто в стандарт и много инструменти предлагат интеграция с нея. Адресът на проекта, от където може да се свали последната версия на библиотеката (текущо 2.2) е http://www.nunit.org.

Какво е автоматизиран unit тест?

Unit тестът е код, който се грижи да постави обектите, които тестваме, в определено състояние и да провери дали нашите очаквания съвпадат с реалността. Тестът има три основни части:

- Подготовка – създаване и инициализиране на тестваните обекти.
- Действие – извикване на методите, чието действие тестваме.
- Проверка – потвърждение, че обектите са в правилното състояние.

Unit тестовете се пишат от програмистите като неделима част от процеса по разработката на кода. По същество, те са тестове от тип "бяла кутия", защото използват знанието за конкретната имплементация. Така се постига максимална ефективност при проверките.

Писането на unit тестове дава няколко съществени предимства при разработката. Тестовете дават лесен начин да се упражни голяма част от кода на приложението за кратко време. Така, след всяка промяна може бързо да се установи дали той работи както очакваме. Дори елементарни и минимални тестове често разкриват проблеми, които не бихме предполо-

жили, че съществуват до последния момент преди планираното публикуване на продукта. Unit тестовете служат като документация, когато ползваме и модифицираме чужд код, демонстрирайки очаквания начин на употребата на класовете и методите. Като втори клиент на кода, тестовете играят ролята и на дизайн инструмент, карайки ни да ограничаваме зависимостите между компонентите и да оформяме по-точни абстракции.

Писане на тестове с Nunit

Да разгледаме в детайли процеса на писане на unit тестове. Той включва някои основни стъпки: създаване на тестови класове и тестови методи, инициализация и почистване на тестовия процес, извършване на серия проверки и др.

Структуриране на кода

Препоръчително е за всеки клас от имплементацията да имаме поне един отделен клас, който да съдържа тестовете. Можем да държим класовете в отделна директория или в отделен проект, така че да ги разделим лесно, когато публикуваме release версия на приложението ни. Най-директният начин да отделим тестовете е да ги сложим в техен собствен проект. От друга страна съхраняването в същия проект, има предимството, че можем да използваме и тестваме **`internal`** класове и методи, които не искаме да са публично достъпни.

Проверки

Проверяването за състоянието на обектите се извършва чрез статичните методи на класа Assert. Чрез тях се извършат стандартни проверки за истинност, равенство, идентичност и др.

```
Assert.AreEqual(2, 1 + 1);
Assert.AreEqual("hello", "hello world".Substring(0, 5),
   "Substring failed!");

Assert.IsTrue(2 > 1);
Assert.IsNotNull(users["Jason"], "User not found.");
```

Всеки метод може да получи като допълнителен параметър низ със съобщение, което се показва, ако проверката пропадне. Това е особено полезно при еднотипни проверки, където не е очевидно коя точно е пропаднала.

Тестови класове и методи

NUnit изисква да маркираме класовете с тестове с атрибута **`TestFixture`**. След като бъде открит тестовия клас в асемблито, се издирват методите му маркирани с атрибута **`Test`** и те подлежат на изпълнение. Да разгледаме един минимален тестов клас:

```
using System;
using NUnit.Framework;

namespace OrderSample.Tests
{
   [TestFixture]
   public class OrderTest
   {
      public OrderTest()
      {
      }

      [Test]
      public void EmptyOrder()
      {
         Order empty = new Order();
         Assert.AreEqual(0, empty.Total);
      }
   }
}
```

Инициализация и почистване

Тестовите класове в NUnit логически представляват постановки. След като се подготви дадено състояние на група обекти могат да бъдат изпълнени действията и проверките. Можем да дефинираме инициализиращата логика в метод, маркиран с атрибута **SetUp**, както и почистващата в друг такъв, маркиран с **TearDown**. Изпълнявайки тестовите методи, NUnit се грижи да извика първо инициализиращия метод преди всеки тест. Същото се прави и за почистващият метод след като теста завърши. По този начин можем да отделим общия за постановката код и да избегнем дубликацията в тестовете. Като развитие на горния пример можем да опишем метод, който да създава поръчка и да я записва преди теста. Като почистваща логика вмъкваме изтриването на тестовата поръчка.

```
using System;
using NUnit.Framework;

namespace OrderSample.Tests
{
   [TestFixture]
   public class OrderTest
   {
      private Order currentOrder;

      public OrderTest()
      {
      }
```

```
    [SetUp]
    public void SetUp()
    {
      currentOrder = new Order();
      currentOrder.Save();
    }

    [TearDown]
    public void TearDown()
    {
      currentOrder.Delete();
    }

    [Test]
    public void EmptyOrder()
    {
      Assert.AreEqual(0, currentOrder.Total);
    }

    [Test]
    public void OneItem()
    {
      OrderItem item = new OrderItem("Bread", 1.5);
      currentOrder.Items.Add(item);
      Assert.AreEqual(1.5, currentOrder.Total);
    }
  }
}
```

NUnit ще създаде и запази две отделни инстанции на Order за двата теста. Аналогично, те ще бъдат изтрити след изпълнението на теста. Това създаване на постановката всеки път осигурява независимостта на всеки тест от останалите.

NUnit предлага възможност да дефинираме глобално инициализиране и почистване за цялата постановка. Това можем да направим, ако маркираме методи с **`TestFixtureSetUp`** и **`TestFixtureTearDown`** атрибутите. Тези методи ще бъдат изпълнени само веднъж за даден клас и са удобни за работа с ресурси, които се използват от всички тестови методи.

Проверки за изключения

Често при тестване на обработката на грешки се налага да проверим дали даден метод хвърля изключение при подадени невалидни данни. В повечето библиотеки това се прави с прихващане на изключението, като в края на **`try`** блока проваляме теста с **`Assert.Fail()`** извикване:

```
[Test]
public void ManualExceptionCheck()
{
```

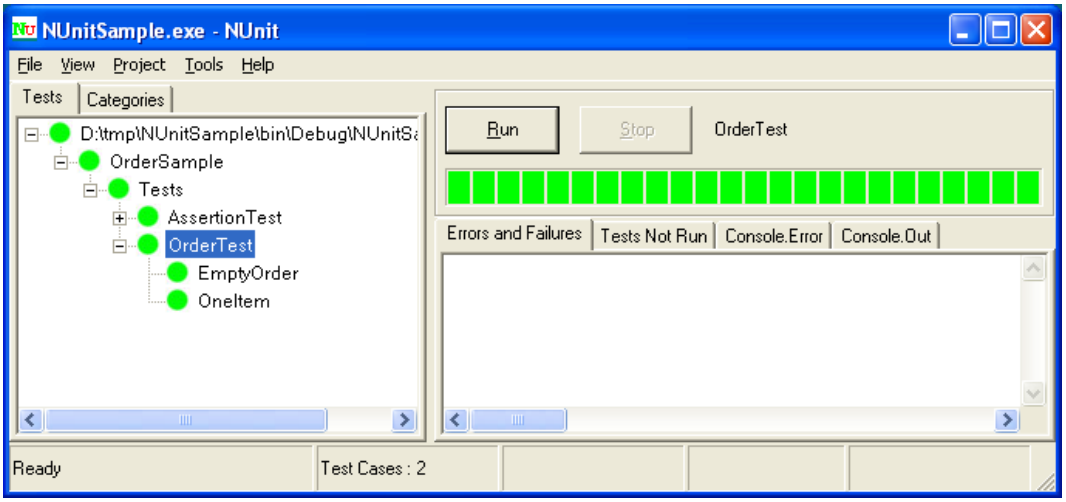
```
   try
   {
      currentOrder.Customer = null;
      Assert.Fail("Null customer should not be allowed");
   }
   catch(ArgumentNullException)
   {
   }
}
```

NUnit ни улеснява в този тип тестове с атрибута за очаквано изключение – **`ExpectedException`**. Той приема като параметри тип на изключението и евентуален низ със съобщението за грешка, което трябва да се съдържа в **`Message`** свойството.

```
[Test]
[ExpectedException(typeof(ArgumentNullException))]
public void ExceptionCheck()
{
   currentOrder.Customer = null;
}
```

Изпълнение на тестовете

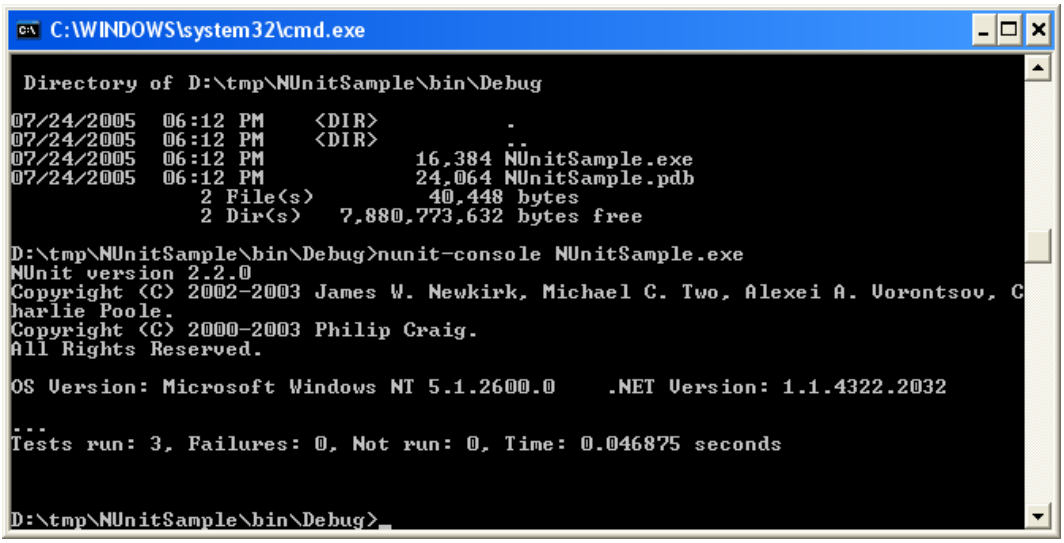
Тестовете са обикновен .NET код и се компилират в някое асембли. NUnit предлага два инструмента за изпълнение, които чрез отражение намират всички тестови класове и ги изпълняват.



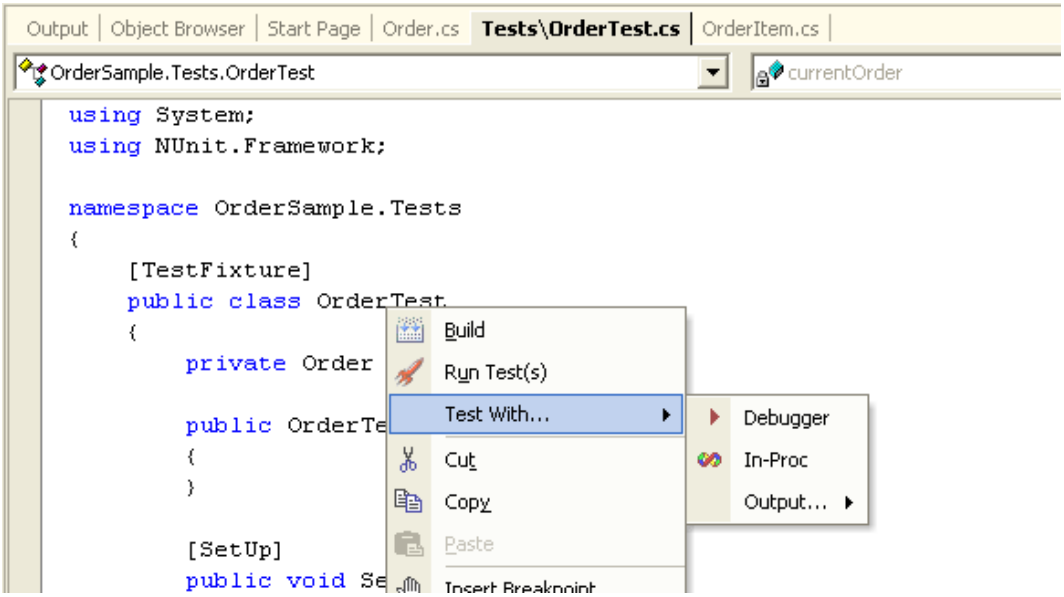
Най-лесен за употреба е инструментът с графичен интерфейс (вж. фигурата по-горе). Той поддържа проекти от тестове, с които можем да укажем няколко асемблита с тестове. Тестовете се показват в дървовидна структура, базирана на пространствата от имена, в които те се намират. Можем да изпълняваме тестовете като ги изберем и щракнем върху бутона

"Run". Резултатът се визуализира веднага след изпълнението. В случая лентата за прогреса е зелена. Тя става червена когато някой тест пропадне.

Графичният интерфейс е удобен, докато разработваме кода и тестовете, но не е достатъчно гъвкав за автоматизиране на изпълнението. За тази цел в NUnit дистрибуцията е включено конзолно приложение, което можем да използваме като му предаваме параметри на командния ред:



Програмистите, които са свикнали да не напускат Microsoft Visual Studio.NET могат да използват добавката TestDriven.NET (http://www.testdriven.net):



Тя позволява директно да изпълним някой тестов метод, всички методи в клас, или всички тестове в проект. Много удобна е и възможността да изпълним тест с дебъгера.

Характеристики на добрите тестове

Добрите тестове могат да предотвратят много дефекти и да ни дадат сигурност, с която да променяме и развиваме имплементацията с по-голяма скорост. Некачествените тестове могат да показват фалшиви тревоги, да са твърде бавни или несигурни – така могат да забавят проекта. Има няколко критерия, които могат да ни помогнат да постигнем положителните ефекти и да избегнем отрицателните. Добрият тест е:

- **Автоматичен** – може да се изпълни с една команда и веднага да разберем дали е успешен или не. Няма нужда от ръчни проверки или инспекции, за да се установи наличието на проблем.
- **Пълен** – покрива всичко, което може да се провали. Всяка част от имплементацията, за която се опасяваме, че може да се провали, трябва да бъде покрита от поне един тест.
- **Повторяем** – две отделни изпълнения трябва да дадат еднакви резултати. Не трябва да има ненужни зависимости от външни компоненти, които не контролираме директно.
- **Независим** – един тест не трябва да е зависим от дейности, които се извършват от друг тест. Всеки тест се грижи за инициализацията и почистването си, като оставя средата в такова състояние, че да не пречи на другите или на самия себе си. Зависимости от реда на изпълнението на тестовете обикновено означават проблем с инициализацията, който трябва да елиминираме. В новата версия на NUnit може да има възможност тестовете да се изпълняват в случаен ред, за да се избегнат такива зависимости.
- **Професионален** – тестовият код не е "второ качество". Обикновено и в него израстват абстракции, които ни улесняват в проверките или настройката на средата. Постоянната грижа за дизайна и почистването на тези абстракции чрез практики като преработка на кода (refactoring) ни гарантира, че ще можем да променяме тестовете със същата скорост както и тестваната имплементация.

Какво да тестваме като програмисти?

Често се сблъскваме с проблема колко време да отделим за тестване на даден клас. Тестването на всички възможни начини да се счупи един клас и доказването, че кодът е непробиваем, често не е практично. Такова пълно осигуряване става излишно, ако приемем, че unit тестовете са просто втори клиент на кода. Те използват компонентите по същия начин както и другите части от кода ни. Това ограничава задачата ни до тестването само на действителните сценарии за употреба. Не е нужно да доказваме, че кодът работи във всички възможни случаи, а само в тези, в

които го използваме. Писането на добрия тест започва с изискванията, които имаме. Ако трябва да напишем клас, който обработва ред от log файл с информация бихме започнали с тест, който покрива основния успешен сценарий:

```
[Test]
public void ParseLine()
{
   LogLine line = new LogLine("INFO: Process Started");
   Assert.AreEqual(LineType.Info, line.Type);
   Assert.IsFalse(line.IsError);
}
```

Следващото изискване е да разберем, дали файлът съдържа невалидни редове. Тестът за проверка дали обработваме добре невалидните префикси за тип на log съобщение би изглеждал така:

```
[Test]
[ExpectedException(typeof(IllegalLogFileException), "Illegal
message type prefix.")]
public void IllegalLine()
{
   LogLine line = new LogLine("ASDFGH: Process Started");
}
```

Най-важното е да покрием изискванията към кода като функционалност. Опитваме се с тестовете да демонстрираме, че кодът прави това, което трябва. Не се стремим формално да доказваме, че не правим това, което не трябва. Когато възникне ново изискване към кода и нов начин на употребата му, трябва да добавим нов тест. Естествено, концентрацията на положителната част от поведението на кода не трябва да се възприема като "розови очила пред очите ни". Трябва да проверяваме с тестове и надеждната обработка за грешки и сигурността, когато те са част от изискванията към приложението ни.

Какво да направим, когато открием дефект в програмата? Дефектите са неразбрани изисквания или недостатъчно добра имплементация. И в двата случая най-добрият подход би бил да добавим тест, който би пропаднал в дадения случай на неправилно поведение. Така, не само променяме кода и поправяме грешката, но и гарантираме, че този тип дефект никога няма да се появи отново. С разрастването на проекта, този вид подсигуряване става все по-важно. Доброто покритие на имплементацията с качествени автоматизирани тестове ни спасява от попадане в печално известната ситуация, където елиминирането на един бъг води до създаването на два нови или "събужда" някой предишен.

Улесняване на тестването

Как да създадем инстанция на клас **A**, като тя използва класове **B** и **C**, а те от своя страна използват други? Как да извикаме метод, който се свързва

с база данни и разчита на предварително въведени данни в няколко таблици? Не винаги има лесни и еднозначни отговори. Прекалено големите зависимости на един компонент от други са белег за лош обектно-ориентиран дизайн. Ако това се случи в контролиран от нас код, можем да го променим, като с улесняването на тестването подобряваме и дизайна. Основният принцип е, че ако обектите са трудни за създаване поотделно и съответно трудни за тестване, то дизайнът на кода има нужда от подобрение. Когато кодът е в някоя библиотека, която не контролираме, единственият изход е да укажем стриктно границите при работата с този код и да дефинираме интерфейси за употреба, чиито имплементации можем да подменяме при тестовете.

Как бихме написали клас, чието поведение се контролира от конфигурационен файл? Нека разгледаме списък с продукти и начина за смятането на общата им цена. Списъкът се съхранява в XML файл, като атрибутите на закупените продукти са цена, отстъпка при промоция и количество. Един първоначален подход към имплементацията би бил да прочетем информацията от файла и да сметнем общата цена в един цикъл. Това, обаче би затруднило тестването на обектите, тъй като ще изисква от нас да подготвим конфигурационния файл преди тестовете и да почистим промените. Твърде честият достъп до файловата система може да забави тестовете и да намали желанието ни да ги пускаме. Затова можем да скрием четенето на продуктите зад интерфейса **`IOrderItemReader`**, което ни позволява да тестваме логиката за отстъпките и сумите отделно в **`OrderItem`** и **`Order`** класовете. За удобство имплементираме **`IOrderItemReader`** в самия тестов клас. Дефинираме **`Read()`** метода, който връща предварително подготвени **`OrderItem`** обекти. В нашия случай това е един продукт с отстъпка от 20%. Тестът проверява дали крайната цена е изчислена правилно с отстъпката:

```
[TestFixture]
public class DiscountedOrderTest : IOrderItemReader
{
   public DiscountedOrderTest()
   {
   }

   [Test]
   public void OneDiscountedItem()
   {
      Order testOrder = new Order(this);
      Assert.AreEqual(4.0, testOrder.Total);
   }
   #region IOrderItemReader Members

   public OrderItem[] Read()
   {
      OrderItem item = new OrderItem("cheese", 5.0);
      item.Discount = 0.20;
```

```
        return new OrderItem[]{item};
    }

    #endregion
}
```

С тази техника можем да разделим отговорностите между няколко обекта и да ги тестваме поотделно. Тестването на истинската имплементация на **IOrderItemReader** също не се нуждае от истински достъп до файлове. Там можем да се доверим на базовите класове от .NET средата и да създадем тестов **Stream** обект в паметта или да заредим **XmlDocument** обект от низ чрез **LoadXml** метода.

Mock обекти (Mock objects)

Намаляването на зависимостите между обектите и разделянето на отговорностите с интерфейси улеснява значително тестването на поведението им. Понякога някои обекти не се контролират директно от нас или се инициализират трудно. Трудната инициализация може да зависи от връзки към база данни или да разчита на показване на някаква форма на графичен потребителски интерфейс. Възможно е това да са компоненти, които се имплементират от друг програмист и още не са завършени. Тестването на обекти, които работят с такива компоненти може да се осъществи чрез създаването на фалшиви имплементации на интерфейсите. Те могат да връщат предефинирани стойности или да следят колко пъти и с какви параметри е извикан даден метод. Ръчното създаване на такива имплементации може да стане досадно и трудоемко. От тук възниква и необходимостта от библиотеки, които да ни улесняват в това. Има два популярни подхода за създаването на mock обекти: генерация на код, който компилираме с тестовете или динамично създаване по време на изпълнение с помощта на класовете от пространството от имена **Reflection.Emit**. Вторият подход е за предпочитане, защото прави тестовете по-лесни за поддръжка.

Работа с NMock

NMock е почти директен пренос за .NET на Java библиотеката за mock обекти **jMock**. Тя ни дава възможност да създадем имплементация на интерфейс или да предефинираме виртуален метод на някой клас по време на изпълнение. Допълнително можем да фиксираме връщаната от метода стойност или да опишем правилата за възможните аргументи, които да получава.

Най-често обектите използват други обекти, за да получат от тях някаква информация. Можем да тестваме имплементацията на някой сценарий, като подадем обект, който винаги да връща избрана от нас стойност - такава, която ще предизвика изпълнението на сценария. Този тип тестови обекти са известни под името "стъб" (stub). NMock ни позволява да

създаваме такива обекти чрез предварително конфигуриране на връщаните от методите стойности. Нека разгледаме тест за клас, който управлява права за достъп на потребители. В теста използваме имплементация на стандартния **System.Security.Principal.IPrincipal** интерфейс:

```
[Test]
public void AllowAdministrators()
{
   DynamicMock principalMock =
     new DynamicMock(typeof(IPrincipal));
   principalMock.SetupResult("IsInRole", true, typeof(string));

   IPrincipal principal = (IPrincipal)principalMock.MockInstance;
   SecurityManager manager = new SecurityManager(principal);
   Assert.IsTrue(manager.AllowResource("Administrator Area"));
}
```

Свойствата на имплементацията се контролират от **DynamicMock** обекта. След като го конфигурираме, от свойството му **MockInstance** можем да получим обект, имплементиращ желания интерфейс. Искаме да проверим, че обектът от тип **SecurityManager** ще даде достъп на администраторите до ресурса "Administrator Area". За целта конфигурираме фалшивата имплементация да връща **true** за метода **IsInRole**.

Друг начин на употреба на mock обектите е като средство да следим взаимодействия. Можем да конфигурираме mock обекта да следи колко пъти е бил извикан някой метод и с какви параметри. Дефиницията на очакваните извиквания се прави с **Expect** методите. Правилата за допустимите параметри се дефинират чрез класовете, имплементиращи **IConstraint**: **IsEqual(…)**, **IsNull()**, **IsIn(…)** и др. Да разгледаме тест, в който **SecurityManager** проверява дали потребителят е в групата "Administrators" и след това дали е в една от двете групи "Backup operators" и "Developers":

```
[Test]
public void BackupPermissions()
{
   DynamicMock principalMock =
     new DynamicMock(typeof(IPrincipal));
   principalMock.ExpectAndReturn("IsInRole", true,
     "Administrators");
   principalMock.ExpectAndReturn("IsInRole", true,
     new IsIn("Backup operators", "Developers"));

   IPrincipal principal = (IPrincipal)principalMock.MockInstance;
   SecurityManager manager = new SecurityManager(principal);
   Assert.IsTrue(manager.AllowResource("Backup"));

   principalMock.Verify();
```

```
}
```

В теста извикваме метода **AllowResource(…)**, като преди това дефинираме очакванията той да извика два пъти **IsInRole** на **IPrincipal** обекта. Първият път ограничаваме възможните параметри до низа "Administrators", а вторият позволяваме като аргумент "Backup operators" или "Developers". Извикването на **Verify()** метода сигнализира приключване на извикването на методи върху фалшивата имплементация. Ако до този момент методът **IsInRole(…)** е извикан само веднъж, тестът ще пропадне с подобно съобщение:

```
TestCase 'OrderSample.Tests.PrincipalTest.BackupPermissions'
failed: NMock.VerifyException : MockIPrincipal.IsInRole() not
called enough times
expected:2
 but was:<1>
   at NMock.Assertion.AssertEquals(String message, Object
expected, Object actual)
   at NMock.Method.Verify()
   at NMock.Mock.Verify()
   d:\tmp\nunitsample\tests\principaltest.cs(41,0): at
OrderSample.Tests.PrincipalTest.BackupPermissions()
```

Разширения на NUnit

NUnit библиотеката предлага базовата функционалност за изпълнение на програмистки тестове. Понякога се налага да работим със специфични библиотеки с по-сложен протокол за достъп до данните на техните обекти. В такива случаи често еволюират набор от класове, които улесняват писането на тестове. За някои стандартни ситуации и библиотеки това вече е направено и може да ни спести сериозни усилия.

NUnitAsp за ASP.NET приложения

NUnitAsp е разширение за NUnit, което симулира потребителските действия, извършвани на една уеб страница. Библиотеката изгражда абстракцията за страница, отворена с уеб браузър и предлага достъп до контролите вътре. За повечето сървърни контроли на ASP.NET имаме вече предефинирани помощни класове, чрез които можем да контролираме приложението. Класовете използват конвенция за именуване образувана от името на сървърния контрол и суфикса **Tester**: **ButtonTester**, **TextBoxTester**, **LabelTester** и др. Всички NUnitAsp тестове трябва да наследяват класа **NUnit.Extensions.Asp.WebFormTestCase**. Заради наследяването не можем да маркираме **SetUp** и **TearDown** методи с обичайните NUnit атрибути и за инициализация и почистване трябва да предефинираме виртуалните методи **SetUp()** и **TearDown()** на базовия клас. Базовият клас също така наследява **WebAssertion** класа и предлага допълнителни

методи за проверка на състоянието (като **AssertEquals()** и **AssertVisibility()**), които е препоръчително да използваме.

Как бихме тествали една страница, която записва информация за потребителя? Страницата може да има поле за име и бутон за запис, който извежда текст с резултата от операцията в един сървърен **Label** контрол. Дефинираме два теста: първият (**SaveDetails**) покрива нормалния сценарий, а във втория (**DontSaveInvalidNames**) опитваме да предотвратим въвеждането на невалидни данни:

```
using System;
using NUnit.Framework;
using NUnit.Extensions.Asp;
using NUnit.Extensions.Asp.AspTester;

namespace PetStore.Tests
{
   [TestFixture]
   public class DetailsTest : WebFormTestCase
   {
      private TextBoxTester nameBox;
      private ButtonTester saveButton;
      private LabelTester messageLabel;

      public DetailsTest()
      {
      }

      protected override void SetUp()
      {
         Browser.GetPage
            ("http://localhost/PetStore/UserDetails.aspx");
         nameBox = new TextBoxTester(
            "nameBox", CurrentWebForm);
         saveButton = new ButtonTester(
            "saveButton", CurrentWebForm);
         messageLabel = new LabelTester(
            "messageLabel", CurrentWebForm);
      }

      [Test]
      public void SaveDetails()
      {
         nameBox.Text = "John Smith";
         saveButton.Click();

         AssertEquals("User details saved successfully.",
            messageLabel.Text);
      }
```

```
    [Test]
    public void DontSaveInvalidNames()
    {
      nameBox.Text = "";
      saveButton.Click();

      AssertEquals("Please enter a valid name.",
        messageLabel.Text);
    }
  }
}
```

В **SetUp()** метода инициализираме страницата и подготвяме **Tester** обектите. За да открием контрола на формата използваме сървърното му свойство ID, което е стандартен механизъм в ASP.NET програмирането.

NUnitForms за WinForms приложения

NUnitForms е по-млад проект от NUnitAsp и е в голяма степен вдъхновен от него. Архитектурата е подобна на тази на NUnitAsp. Отново имаме набор от **Tester** класове за стандартните контроли. Няма нужда да наследяваме други класове, тъй като при WinForms имаме по-голям контрол над приложението. Единствената особеност е, че тестовете "намират" контролите по **Name** свойството им и името на формата. Ако имаме само една форма можем да изпуснем името – така опростяваме кода. За да остане формата само една, можем да я инициализираме в **SetUp()** метода и да я затваряме в **TearDown()**. Ето и тест за приложение, запазващо информация за потребителя, подобно на предишния пример:

```
using System;
using NUnit.Framework;
using NUnit.Extensions.Forms;
using PetStore;

namespace Petstore.Tests
{
  [TestFixture]
  public class DetailsTest
  {
    private DetailsForm form;
    private LabelTester messageLabel;
    private TextBoxTester customerName;
    private ButtonTester saveButton;

    public DetailsTest()
    {
    }

    [SetUp]
```

```
    public void SetUp()
    {
      form = new DetailsForm();
      form.Show();

      messageLabel = new LabelTester("messageLabel");
      customerName = new TextBoxTester("nameBox");
      saveButton = new ButtonTester("saveButton");
    }

    [TearDown]
    public void TearDown()
    {
      form.Close();
    }

    [Test]
    public void SaveUserInfo()
    {
      customerName.Enter("John Smith");
      saveButton.Click();

      Assert.AreEqual("Details saved successfully.",
        messageLabel.Text);
    }

    [Test]
    public void DontAllowEmptyNames()
    {
      customerName.Enter(string.Empty);
      saveButton.Click();

      Assert.AreEqual("Please enter a valid name.",
        messageLabel.Text);
    }
  }
}
```

Използвана литература

- Andrew Hunt, David Thomas, Pragmatic Unit Testing In C# with NUnit
- Kent Beck, Test Driven Development: By Example
- Ronald E. Jeffries - Extreme Programming Adventures in C#
- NUnit Documentation - http://nunit.org/documentation.html
- NMock overview – http://www.nmock.org
- Mock Objects Web site – http://www.mockobjects.com

- Martin Fowler, Mocks Aren't Stubs - http://www.martinfowler.com/articles/mocksArentStubs.html
- NUnitAsp Documentation - http://nunitasp.sourceforge.net/documentation.html
- NUnitAsp API Reference - http://nunitasp.sourceforge.net/api.html
- NUnitForms Documentation - http://nunitforms.sourceforge.net/docs.html
- NUnitForms API Reference - http://nunitforms.sourceforge.net/MSDN/index.html

Log4net

Log4net е библиотека с отворен код за извеждане на лог (log) съобщения. Тя е наследник на изключително успешната разработка за Java log4j. Тази популярна, доказана архитектура има реализации на повече от десет програмни езика. Log4net (текущо версия 1.2) е имплементацията за .NET и може да бъде намерена на адрес: http://logging.apache.org/log4net/.

За техниката "логинг"

Генерирането на лог съобщения в кода, известно като логинг (logging), представлява проста техника за изследване вътрешното поведение на кода. Тази практика се ползва за разработката на всякакви приложения, но полезността й проличава в най-голяма степен при многонишкови или разпределени системи. Понякога използването на логинг може да е единственото налично средство за изследване на проблеми, примерно, когато след внедряване на приложението нямаме достъп до дебъгер.

Опитът показва, че записването на лог съобщения е важен елемент в разработката. Едно от нещата, което го отличава от техниката на дебъгване, е това, че изходът му може да бъде съхраняван и по-късно анализиран. Веднъж заложено в кода, генерирането на логинг съобщения се извършва повторяемо без човешка намеса. Както по време на разработка, така и след внедряване, използването на логинг може да ни спести много време в диагностицирането на проблеми и справянето с тях.

Предизвикателствата пред log4net

Използването на логинг, освен изброените ползи, носи и някои рискове:

- Усвояването на добрите практики за логинг изисква известни усилия. Една невнимателна реализация може да доведе до неочаквани странични ефекти.
- Лог съобщенията могат да намалят бързодействието на приложението.
- Ако се генерира твърде много логинг информация, преглеждащият може да се загуби в нея.

За да бъдат избегнати тези опасности, log4net е проектирана да бъде лесно разбираема, надеждна, бърза и конфигурируема. Основните характеристики на архитектурата ѝ включват:

- Лог съобщенията, които се генерират в кода, влияят минимално върху бързодействието на приложението.
- Можем селективно да контролираме кои лог съобщения да бъдат извеждани, в какъв вид и къде.
- Лесно можем да извеждаме лог съобщения към множество различни цели: файл, база от данни, конзола и други.
- Чрез използването на конфигурационни файлове можем динамично да настройваме логинг процес по време на изпълнение на приложението.

Компоненти на log4net

Log4net има три основни компоненти: логери (**loggers**), апендери (**appenders**) и оформления (**layouts**). С тяхна помощ разработчиците могат да логват съобщения в зависимост от типа и нивото им, както и да контролират в какъв формат и къде да бъдат записани. Тези компоненти могат да бъдат дефинирани директно в кода, или под формата на XML файл. В демонстрационния пример по-долу ще използваме гъвкав подход, разчитащ на XML конфигурация.

Йерархия на логерите

Отличителна характеристика на всички сериозни логинг библиотеки е възможността за избирателно активиране на това кои лог изрази да бъдат обработени. За целта log4net предлага мощни механизми за категоризация на събитията, които разработчиците да използват съгласно нуждите на приложението.

За да позволи гъвкаво контролиране на различните аспекти на логинг процеса, log4net въвежда концепцията за логери. Това са обекти, които се организират в йерархия, използвайки схема на именуване, подобна на пространствата от имена в .NET. Примерно логер с име **`"Foo.Bar"`** се счита за родител на логер **`"Foo.Bar.Baz"`**. На върха винаги стои един базов (**`root`**) логер. Подобно на йерархиите на класове в ООП, логерите наследяват характеристиките на предшествениците си, като могат да добавят нови или да предефинират някои от тях.

Често използвана стратегия за моделирането на логерите в едно приложение е дефинирането на логер за всеки от класовете и именуването му с пълното име на класа. По този начин става ясно откъде произлиза всяко събитие, а и при създаване на йерархията на логери се използва наготово дизайна на компонентите.

Нива на логерите

На всеки логер може да му бъде зададено ниво. Ако не му е указано изрично, той използва нивото на най-близкия си предшественик. Log4net дефинира следните нива на логинг (наредбата е по критичност): **ALL < DEBUG < INFO < WARN < ERROR < FATAL < OFF**. Всяка заявка за логинг се активира само, ако нивото ѝ е по-голямо или равно на нивото на нейния логер. В противен случай тя бива игнорирана.

Работа с логери

Достъпът до инстанция на логер става през статичния метод **log4net.LogManager.GetLogger(…)**, връщащ интерфейс от тип **log4net.ILog**. Функцията приема като единствен аргумент или низ (името на логера) или **System.Type** (което е подходящо в случаите, когато името на логера съвпада с това на класа):

```
ILog log = LogManager.GetLogger("LoggingExample.User");
```

Интерфейсът **ILog** предоставя следните методи и свойства (за краткост са изброени само тези с **DEBUG**, като тези за **INFO**, **WARN**, **ERROR** и **FATAL** са аналогични):

```
// DEBUG properties and methods
bool IsDebugEnabled { get; }
void Debug(object message);
void Debug(object message, Exception t);
void DebugFormat(string format, params object[] args);
void DebugFormat(IFormatProvider provider, string format,
  params object[] args);
```

Заявките за логинг се правят с извикване на методите **Debug(…)**, **Info(…)**, **Warn(…)**, **Error(…)** и **Fatal(…)** върху инстанция на **log4net.ILog**:

```
log.Info("Database connection established successfully.");
```

Имената на горните функции определят нивото на логинг събитието, т. е. **log.Info("…")** е заявка за логинг с ниво **INFO**.

Апендери (Appender)

Log4net позволява лесен логинг към множество цели чрез концепцията за апендери, които представляват компоненти за показване или съхраняване на съобщения. Към един логер могат да бъдат прикачени един или повече апендери. Всяко събитие за даден логер бива пращано към всичките му дефинирани апендери и към тези, асоциирани с по-високо стоящите в йерархията логери. Log4net идва със следните предварително дефинирани стандартни апендери:

Тип	Описание
AdoNetAppender	Записва логинг събития в база от данни, използвайки подготвени SQL изрази или съхранени процедури.
AnsiColorTerminalAppender	Записва оцветени логинг събития в ANSI терминален прозорец.
AspNetTraceAppender	Записва логинг събития в ASP трасиращ контекст. Те могат да бъдат показани като част от ASP страниците или в ASP трасираща страница.
ColoredConsoleAppender	Записва оцветени логинг събития в Windows конзолата на приложенията.
ConsoleAppender	Записва логинг събития в конзолата на приложението, в стандартния изход или в стандартния изход за грешки.
EventLogAppender	Записва логинг събития в Windows Event Log.
FileAppender	Записва логинг събития във файл.
LocalSyslogAppender	Записва логинг събития в локалния syslog сервиз (само за UNIX / Linux).
MemoryAppender	Запазва логинг събития в буфер в паметта.
NetSendAppender	Записва логинг събития в Windows Messenger сервиз. Тези съобщения се показват в диалогов прозорец.
OutputDebugStringAppender	Записва логинг събития в дебъгера. Ако приложението няма свой дебъгер, но е активен системният дебъгер, той показва текста.
RemoteSyslogAppender	Записва логинг събития към отдалечен syslog сервиз чрез UDP пакети.
RemotingAppender	Записва логинг събития към отдалечена цел използвайки .NET remoting.
RollingFileAppender	Записва логинг събития във файловата система. Може да бъде конфигуриран да използва няколко файла в зависимост от ограничения за дата и размер на файла.
SmtpAppender	Изпраща логинг събития към зададен имейл адрес.

TelnetAppender	Клиентите се свързват с Telnet, за да получат логинг събития.
TraceAppender	Записва логинг събития в стандартната трасировъчна система на .NET.
UdpAppender	Изпраща логинг събития като UDP пакети към отдалечена точка или multicast група, използвайки **UdpClient**.

Филтри

В log4net могат да бъдат указани филтри за по-детайлен контрол на това кои съобщения да преминават през различните апендери. Често ползвани са филтри от тип праг (пределно ниво). При тях само събитията с ниво, равно или по-голямо на указаната стойност, ще бъдат логвани чрез съответния апендер. Могат да бъдат дефинирани и по-сложни операции чрез последователности от филтри. Вградените в log4net филтри са:

Тип	Описание
DenyAllFilter	Игнорира всички логинг събития.
LevelMatchFilter	Точно съвпадение с нивото на събитието.
LevelRangeFilter	Попадане на нивото на събитието в определен диапазон.
LoggerMatchFilter	Съвпадащо начало на името на логер.
PropertyFilter	Съвпадение с подниз на стойност на свойство.
StringMatchFilter	Съдържане на подниз в текста на събитието.

Оформления (Layouts)

Когато искаме да настроим не само целите на логинг, а и изходния формат на съобщенията, трябва да асоцииираме оформление към даден апендер. Така можем да форматираме генерирания текст спрямо нуждите ни, преди той да бъде записан. Например чрез **PatternLayout** можем да укажем изходен формат по начин подобен на познатата от C функция **printf(…)**. Като използваме комбинацията от спецификатори като **%timestamp**, **%thread**, **%level**, **%logger**, **%message**, **%newline** и други ще получим като изход текста, оформен в желания от нас вид. В log4net са включени следните оформления:

Тип	Описание
ExceptionLayout	Показване на текста на изключението в логинг събитието.
PatternLayout	Форматиране на логинг събитието спрямо набор от спецификатори.

RawTimeStampLayout	Извличане на времето от логинг събитието.
RawUtcTimeStampLayout	Извличане на времето от логинг събитието в Universal Time формат.
SimpleLayout	Опростено форматиране на логинг събитието: **[level] - [message].**
XmlLayout	XML форматиране на логинг събитието.
XmlLayoutSchemaLog4j	XML форматиране на логинг събитието по съответна на log4j DTD схема.

Други характеристики на log4net

Нека разгледаме някои опции, които ни предлага log4net в допълнение към основната функционалност.

Поддръжка на множество платформи

Log4net поддържа следните платформи:

- Microsoft .NET Framework 1.0 (1.0.3705)
- Microsoft .NET Framework 1.1 (1.1.4322)
- Microsoft .NET Compact Framework 1.0 (1.0.5000)
- Mono 1.0
- Microsoft Shared Source CLI 1.0
- CLI 1.0 Compatible

Динамична XML конфигурация

Log4net използва XML конфигурационни файлове. Данните могат да бъдат съхранявани като отделен файл или да бъдат вмъкнати като секция в други XML файлове (примерно в **.config** файла на приложението). В XML конфигурацията могат да се дефинират апендери, оформления, нива на логинг и други параметри.

Повечето настройки подлежат на динамична конфигурация, т.е log4net може да наблюдава файла за извършени промени и да ги прилага по време на изпълнение на програмата. Така става възможно диагностициране на проблеми, без да бъде спирано приложението. За системи, които вече са били внедрени и са работещи, това понякога е важно изискване.

Като алтернативен вариант, log4net може да бъде конфигурирана и директно в кода, но така се изгубва възможността за динамични настройки.

Контекстна информация при логинг

Полезна практика е добавянето на допълнителна, контекстно-зависима информация при логинг на съобщения. За целта log4net предлага обек-

тите **GlobalContext** (за глобален контекст) и **ThreadContext** (контекст на нишка). Чрез тях приложението може удобно да съхранява и впоследствие да прикачва към логинг съобщенията данни за контекста на изпълнение. Примерно, в една уеб услуга, след като викащият се автентикира с потребителското си име, то може да бъде запомнено като свойство в **ThreadContext** и след това автоматично да бъде извеждано като част от всички логинг съобщения.

log4net – пример

За да илюстрираме възможностите на log4net сме подготвили следния демонстрационен пример. В него илюстрираме как се използват базовите функции за логинг в едно конзолно приложение **ConsoleAppLog4Net**, използващо XML конфигурация:

```
using System;
using log4net;

// Configure log4net using the .config file
[assembly : log4net.Config.XmlConfigurator()]

namespace ConsoleAppLog4Net
{
   internal class LoggingExample
   {
      private static ILog log = LogManager.GetLogger(
         typeof (LoggingExample));

      private static void Main()
      {
         // Log an info level message
         if (log.IsInfoEnabled)
         {
            log.Info("Application [ConsoleAppLog4Net] Start");
         }

         // Log a debug message. Test if debug is enabled before
         // attempting to log the message. This is not required
         // but can make running without logging faster.
         if (log.IsDebugEnabled)
         {
            log.Debug("This is a debug message");
         }

         try
         {
            Bar();
         }
         catch (Exception ex)
         {
```

```
            // Log an error with an exception
            log.Error("Exception thrown from method Bar", ex);
        }

        log.Error("Hey this is an error!");

        if (log.IsInfoEnabled)
        {
            log.Info("Application [ConsoleAppLog4Net] End");
        }

        Console.Write("Press Enter to exit...");
        Console.ReadLine();
    }

    private static void Bar()
    {
        Goo();
    }

    private static void Foo()
    {
        throw new Exception("This is an Exception");
    }

    private static void Goo()
    {
        try
        {
            Foo();
        }
        catch (Exception ex)
        {
            throw new ArithmeticException("Failed in Goo. " +
                "Calling Foo. Inner Exception provided", ex);
        }
    }
  }
}
```

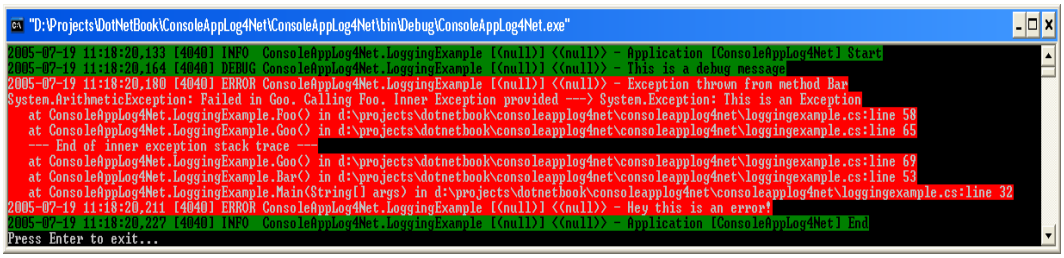
Редът **`[assembly : log4net.Config.XmlConfigurator()]`** показва как можем да заредим конфигурацията на log4net, ако се съхранява в **`.config`** файла на приложението. Конфигурационният файл изглежда така:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <!-- Register a section handler for the log4net section -->
  <configSections>
    <section name="log4net"
      type="System.Configuration.IgnoreSectionHandler" />
```

```
  </configSections>
  <!-- This section contains the log4net config settings -->
  <log4net>
    <!-- Define some output appenders -->
    <appender name="RollingLogFileAppender"
         type="log4net.Appender.RollingFileAppender">
      <file value="rolling-log.txt" />
      <appendToFile value="true" />
      <maxSizeRollBackups value="10" />
      <maximumFileSize value="100" />
      <rollingStyle value="Size" />
      <staticLogFileName value="true" />
      <layout type="log4net.Layout.PatternLayout">
        <header value="[Header]&#13;&#10;" />
        <footer value="[Footer]&#13;&#10;" />
        <conversionPattern value="%date [%thread] %-5level
%logger [%ndc] - %message%newline" />
      </layout>
    </appender>
    <appender name="LogFileAppender"
         type="log4net.Appender.FileAppender">
      <file value="log-file.txt" />
      <appendToFile value="true" />
      <layout type="log4net.Layout.PatternLayout">
        <header value="[Header]&#13;&#10;" />
        <footer value="[Footer]&#13;&#10;" />
        <conversionPattern value="%date [%thread] %-5level
%logger [%ndc] &lt;%property{auth}&gt; - %message%newline" />
      </layout>
    </appender>
    <appender name="ConsoleAppender"
         type="log4net.Appender.ConsoleAppender">
      <mapping>
        <level value="ERROR" />
        <foreColor value="White" />
        <backColor value="Red, HighIntensity" />
      </mapping>
      <mapping>
        <level value="DEBUG" />
        <backColor value="Green" />
      </mapping>
      <layout type="log4net.Layout.PatternLayout">
        <conversionPattern value="%date [%thread] %-5level
%logger [%ndc] &lt;%property{auth}&gt; - %message%newline" />
      </layout>
    </appender>
    <!-- Setup the root category, add the appenders and set the
      default level -->
    <root>
      <level value="WARN" />
```

```
        <appender-ref ref="LogFileAppender" />
        <appender-ref ref="ConsoleAppender" />
      </root>
      <!-- Specify the level for some specific categories -->
      <logger name="ConsoleAppLog4Net.LoggingExample">
        <level value="ALL" />
        <appender-ref ref="RollingLogFileAppender" />
      </logger>
    </log4net>
</configuration>
```

В този XML файл сме дефинирали 3 типа апендери, които използват различни оформления, и сме им задали различни нива на логинг. След като стартираме приложението в конзолния прозорец се визуализира очаквания изход. Лесно отличаваме информационните съобщения, които са със зелен фон, от тези с ниво **ERROR**, които са с червен:



А ето какво е съдържанието на лог файла:

log-file.txt

```
[Header]
2005-07-19 11:22:04,364 [2496] INFO
ConsoleAppLog4Net.LoggingExample [(null)] <(null)> - Application
[ConsoleAppLog4Net] Start
2005-07-19 11:22:04,410 [2496] DEBUG
ConsoleAppLog4Net.LoggingExample [(null)] <(null)> - This is a
debug message
2005-07-19 11:22:04,426 [2496] ERROR
ConsoleAppLog4Net.LoggingExample [(null)] <(null)> - Exception
thrown from method Bar
System.ArithmeticException: Failed in Goo. Calling Foo. Inner
Exception provided ---> System.Exception: This is an Exception
   at ConsoleAppLog4Net.LoggingExample.Foo() in
d:\projects\dotnetbook\consoleapplog4net\consoleapplog4net\loggi
ngexample.cs:line 58
   at ConsoleAppLog4Net.LoggingExample.Goo() in
d:\projects\dotnetbook\consoleapplog4net\consoleapplog4net\loggi
ngexample.cs:line 65
   --- End of inner exception stack trace ---
   at ConsoleAppLog4Net.LoggingExample.Goo() in
```

```
d:\projects\dotnetbook\consoleapplog4net\consoleapplog4net\loggi
ngexample.cs:line 69
   at ConsoleAppLog4Net.LoggingExample.Bar() in
d:\projects\dotnetbook\consoleapplog4net\consoleapplog4net\loggi
ngexample.cs:line 53
   at ConsoleAppLog4Net.LoggingExample.Main(String[] args) in
d:\projects\dotnetbook\consoleapplog4net\consoleapplog4net\loggi
ngexample.cs:line 32
2005-07-19 11:22:04,457 [2496] ERROR
ConsoleAppLog4Net.LoggingExample [(null)] <(null)> - Hey this is
an error!
2005-07-19 11:22:04,473 [2496] INFO
ConsoleAppLog4Net.LoggingExample [(null)] <(null)> - Application
[ConsoleAppLog4Net] End
```

Както виждаме и при такъв малък пример се генерира значителна по обем логинг информация. В едно реално приложение бихме се ориентирали много трудно в лог файловете, ако не използваме възможностите за селективен изход чрез настройки на логерите.

Използвана литература

- Log4net Features - http://logging.apache.org/log4net/release/features.html
- Log4net Manual Introduction - http://logging.apache.org/log4net/release/manual/introduction.html

NHibernate

NHibernate е библиотека за извличане и записване на данните на обекти в релационни бази от данни (object persistence). Тя е .NET имплементацията на изключително популярния инструмент в Java програмирането Hibernate (http://www.hibernate.org/).

Текущата версия на NHibernate е 1.0, но продуктът е стабилен и се ползва в множество реални проекти. Възможностите на тази версия са същите като на Hibernate 2.1. NHibernate е проект с отворен код и може да бъде намерен на http://www.nhibernate.org/.

Взаимодействие между обекти и релационни СУБД

При разработката на съвременни приложения в голяма степен се е наложило използването на обектно-ориентирани подходи. Много приложения се нуждаят от начин да запазват и съответно да извличат данни за обектите, с които оперират. Релационните СУБД са популярно средство за целта, защото предоставят надеждност, ефективност и други желани характеристики при достъпа и манипулирането на данни.

При срещата на обектно-ориентираната парадигма с тази на релационните бази от данни, възникват известни технически предизвикателства. Знанията на разработчиците по ООП не са приложими в реализацията на съхраняването и обновяването на данните чрез традиционните операции за избор, добавяне, промяна и изтриване в базата от данни. Те трябва да мислят в термините на таблици, колони и релации. От тях се изисква добро познаване на езика за манипулиране на релационни данни SQL, който при това се среща в множество форми (диалекти) при различните СУБД.

Моделирането на проблемната област чрез обекти включва както данните, така и логика за обработката им. При проектирането на релационни БД фокусът е единствено върху данните. В резултат на тези различни гледни точки на много приложения се налага да работят едновременно с два модела, различаващи се помежду си в редица аспекти. В ООП боравим с класове, свойства, методи, а в базите от данни работим с понятия като таблици, колони, релации. Допълнително, типизацията на данните е различна и няма еднозначно съответствие между типовете, използвани в програмните езици (и в частност дефинираните в Common Type System), и типовете от базите от данни.

ADO.NET и силно типизирани DataSets

ADO.NET предоставя на .NET програмистите богат на възможности интерфейс за работа с релационни бази от данни. Класовете **`DataSet`**, **`DataTable`**, **`DataRow`** и т.н. дават обектно-ориентиран начин за достъп до данните. Тъй като тези типове представят релационния модел, то възможностите им са ограничени до експресивните характеристики на модела. Техните функции се свеждат до навигация, избор и обновяване на данните. В **`DataSets`** не може лесно да бъде енкапсулирано поведение и логика за по-сложни обработки.

Силно типизираните (strongly-typed) **`DataSets`** са още една стъпка в посока обектно-ориентиран достъп до данните. Тяхната структура се описва чрез XML, от който автоматично се генерира код (чрез инструмента **`xsd.exe`**). Създадените класове са наследници на **`DataSet`**, но в тях достъпът до данните може да се извършва директно през свойства, а не чрез имена на колони. До голяма степен, това решава и проблемите с различията в типизацията на данните, както и се адресират традиционните трудности при представяне на NULL стойностите. Силно типизираните **`DataSets`**, представящи няколко свързани таблици, предоставят допълнително удобни механизми за навигация по релациите.

Програмирането със силно типизирани **`DataSets`** притежава много от характеристиките на обектно-ориентирания код, но обикновено те са продукт на чисто релационно мислене и моделиране. Силно типизираните **`DataSets`** предоставят удобен начин за работа с извадки от базата от данни, но рядко отговарят на всички желани характеристики на обектно-ориентирания дизайн.

Обектно-релационен преход

В практиката се ползват няколко различни подхода, позволяващи прехвърлянето на данни между обектите и базата от данни по начин, избягващ прекомерната обвързаност между тях. NHibernate е представител на популярните решения от тип Object/Relational Mappers (накратко ORM). Подходът за моделиране, познат под името Data Mapper, предоставя независимата услуга за съхраняване на обектите. Там се използва превод на концепциите на релационното моделиране в термините на ООП чрез съответствия от вида клас ↔ таблица и свойство ↔ колона.

Обектно-релационни съответствия

В основата на ORM технологията е дефинирането на съответствия (mappings) между класове и таблици. Данните, използвани в един клас могат да бъдат съхранявани в една или повече таблици. NHibernate поддържа всички често срещани сценарии за връзки между класове и таблици.

След като са дефинирани съответствията, ORM библиотеката поема отговорността да синхронизира обектите от паметта и базата от данни. Различните имплементации използват различни методи за дефиниране на съответствия. В NHibernate те се задават чрез утвърдената технология на XML файлове. Такъв начин на конфигурация има редица предимства пред задаване на съответствията директно в кода, било то декларативно (чрез .NET атрибути) или императивно.

Транзакции

Приложенията често използват бизнес транзакции за осъществяването на някаква работа. В рамките на една транзакция щом бъде променено състоянието на обектите това трябва да се отрази в базата от данни. Някои ORM инструменти, като NHibernate, поддържат списък на обектите с променено състояние в рамките на една транзакция. Това прави възможен ефективния запис в базата от данни, както и справянето с проблеми свързани с конкурентен достъп. Единицата за работа при NHibernate е сесия и се представя с обект от тип **`Session`**.

Синхронизиране на промените

След използване на обектите и промяна на техните данни, възниква въпросът как да се разбере кои данни трябва да бъдат обновени в базата от данни. ORM решенията имат различни подходи за този проблем. Един вариант е разработчикът изрично да отбелязва кога обектите са променени. Алтернативен начин е непосредствено преди обновяване да се сравнят с допълнително обръщение текущите данни с тези, съхранени в базата от данни. NHibernate използва по-добра стратегия, поддържайки за всяка транзакция кеш на състоянията на обектите, участващи в нея. Така библиотеката може автоматично да разбере дали са били извършени промени, изискващи обновяване в базата от данни.

Отложено зареждане (lazy loading)

Обикновено един Data Mapper извлича наведнъж всички данни за обектите, за които отговаря. Понякога обекти от даден тип съдържат голяма йерархия от обекти, които не бихме искали да зареждаме при всяко извличане на данни. За целта NHibernate поддържа отложено зареждане (lazy loading). С този подход данните се извличат тогава, когато станат наистина нужни.

Кеширане

В натоварена среда, примерно при уеб приложения, е препоръчително да се прилага кеширане на обектите. Има различни начини за това, някои са на ниво сесия, други на ниво приложение. NHibernate поддържа кеширане в рамките на един **`Session`** обект.

Език за заявки

Често се налага извличане на обекти по критерии, свързани със стойности на едно или няколко техни свойства. Обикновено достъпът по уникален идентификатор не е достатъчен. Примерно, ако търсим обекти от тип автомобил, ще искаме да зададем условия за модел и цена. За целта NHibernate използва език подобен на SQL – Hibernate Query Language (HQL). Той е доста добре развит и поддържа повечето от концепциите, използвани в съвременните СУБД.

Други функции

ORM инструментите предлагат множество други възможности, но обикновено нуждите на проектите изискват ползването на малка част от тях. NHibernate е сред тези решения, които предоставят богата функционалност и максимална гъвкавост. Широкият обхват на възможностите на библиотеката се предоставя в на пръв поглед огромен програмен интерфейс, но в повечето приложения се налага да използваме негово малко подмножество. В примера по-долу ще се спрем само на най-базовите познания, които са нужни, за започване на работа с NHibernate.

Демонстрационен пример с NHibernate

В примера ще демонстрираме един прост сценарий на използване на Nhibernate, като минем през следните стъпки:

1. Създаваме таблица, в която ще бъдат съхранявани данните за един .NET клас.
2. Създаваме .NET класа.
3. Създаваме файл със съответствията, указващ как NHibernate да извлича и записва в таблицата стойностите на свойствата на класа.
4. Създаваме конфигурационен файл, указващ как NHibernate да се свързва с базата от данни.

5. Използваме функциите, които предоставя NHibernate.

Стъпка 1: Създаване на таблицата чрез SQL

Да си представим, че разработваме проста подсистема за работа с потребителите на уебсайт. Нека създадем таблица **Users**, която има следния вид:

```
CREATE TABLE Users (
  LogonID nvarchar(20) NOT NULL default '0',
  Name nvarchar(40) default NULL,
  Password nvarchar(20) default NULL,
  EmailAddress nvarchar(40) default NULL,
  LastLogon datetime default NULL,
  PRIMARY KEY (LogonID)
)
```

Примерът е върху MS SQL Server 2000, но няма пречки да използваме всяка друга СУБД. NHibernate поддържа всички бази от данни, за които съществува .NET Data Provider.

Таблицата за потребителите ще съдържа стандартни данни: потребителско име, име, парола, e-mail адрес и дата на последно посещение. Сега да напишем .NET клас със съответните свойства.

Стъпка 2: Създаване на .NET клас

Нужен ни е начин да представим потребителите в паметта и да можем да извличаме, променяме и обновяваме данните им. Ще дефинираме клас, който да съответства на таблицата **Users** от базата данни. Ще добавим свойства за полетата на този клас. NHibernate вътрешно ще работи с нашия клас чрез техниката на отражение. Ето как изглежда класът **User**:

```
namespace NHibernate.Examples.QuickStart
{
   public class User
   {
      private string mId;
      private string mUserName;
      private string mPassword;
      private string mEmailAddress;
      private DateTime mLastLogon;

      public User()
      {
      }

      public string Id
      {
         get { return mId; }
         set { mId = value; }
```

```
        }

        public string UserName
        {
            get { return mUserName; }
            set { mUserName = value; }
        }

        public string Password
        {
            get { return mPassword; }
            set { mPassword = value; }
        }

        public string EmailAddress
        {
            get { return mEmailAddress; }
            set { mEmailAddress = value; }
        }

        public DateTime LastLogon
        {
            get { return mLastLogon; }
            set { mLastLogon = value; }
        }
    }
}
```

В примера сме дефинирали свойствата и конструктора на класа с публичен достъп, но NHibernate може да работи както с **public**, така и с **protected**, **internal** и дори с **private** свойства.

Стъпка 3: Създаване на файл със съответствията

Сега ще направим връзката между SQL таблицата и .NET класа ни. За целта трябва да подготвим файл със съответствията (mappings). Препоръчителното е да поддържаме по един файл за всеки клас, да го именуваме по схемата **ClassName.hbm.xml** и да го съхраняваме в същата директория като класа. Ако използваме Visual Studio.NET проект трябва да добавим в него файла като вграден ресурс (embedded resource), за да бъде част от асемблито. Ето как може да изглежда **User.hbm.xml**:

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.0">
  <class name="NHibernate.Examples.QuickStart.User,
       NHibernate.Examples" table="Users">
    <id name="Id" column="LogonId" type="String" length="20">
      <generator class="assigned" />
    </id>
    <property name="UserName" column="Name" type="String"
```

```
        length="40"/>
      <property name="Password" type="String" length="20"/>
      <property name="EmailAddress" type="String" length="40"/>
      <property name="LastLogon" type="DateTime"/>
    </class>
</hibernate-mapping>
```

При дефиниране на съответствия между таблица и клас освен пълното име на класа, трябва да укажем и в кое асембли се намира, за да може NHibernate да го открие и зареди. В случая асемблито е с име **`NHibernate.Examples`** и дори да не укажем дали е **`.exe`** или **`.dll`**, то ще бъде открито.

Таговете **`property`** вършат основната работа за указване на съответствията на ниво колони. Атрибутът **`name`** е за свойството на класа. После следва името на колоната в базата от данни, която дори може да се пропусне, в случай че името й съвпада с това на свойството. Атрибутът **`type`** също не е задължителен - NHibernate ще използва отражение, за да се опита да го познае.

Тагът **`id`** е за първичния ключ на таблицата. По атрибути прилича много на тага **`property`**. Вложеният таг **`generator`** казва на NHibernate как да генерира първичния ключ. Поддържат се множество типове генератори на идентификатор, но в нашия случай, обектът ще ползва ключ, генериран от базата от данни.

Стъпка 4: Създаване на конфигурационен файл за базата от данни

Все още не сме указали коя е базата от данни. Най-лесният начин за това е да предоставим на NHibernate конфигурационна секция в **`.config`** файла на приложението. Ето как може да изглежда тя:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section
      name="nhibernate"
      type="System.Configuration.NameValueSectionHandler,
System, Version=1.0.5000.0,Culture=neutral,
PublicKeyToken=b77a5c561934e089"
    />
  </configSections>

  <nhibernate>
    <add
      key="hibernate.connection.provider"
      value="NHibernate.Connection.DriverConnectionProvider"
    />
    <add
      key="hibernate.dialect"
```

```
      value="NHibernate.Dialect.MsSql2000Dialect"
    />
    <add
      key="hibernate.connection.driver_class"
      value="NHibernate.Driver.SqlClientDriver"
    />
    <add
      key="hibernate.connection.connection_string"
      value="Server=localhost;initial
catalog=nhibernate;Integrated Security=SSPI"
    />
  </nhibernate>
```

Примерът използва **`SqlClient`** драйвер за свързване с база от данни с име **`nhibernate`** на **`localhost`**. Предлагат се и още няколко характеристики, с които да настроим фино как да се осъществява достъпа до базата.

Стъпка 5: Същинската работа с NHibernate

Сега остава да реализираме същинската функционалност, използваща възможностите на NHibernate. За целта добавяме референция към **`NHibernate.dll`** в проекта ни и реализираме следния код:

```
Configuration cfg = new Configuration();
cfg.AddAssembly("NHibernate.Examples");

// Open DB session and start a transaction
ISessionFactory factory = cfg.BuildSessionFactory();
ISession session = factory.OpenSession();
ITransaction transaction = session.BeginTransaction();

// Create new user
User newUser = new User();
newUser.Id = "joe_cool";
newUser.UserName = "Joseph Cool";
newUser.Password = "abc123";
newUser.EmailAddress = "joe@cool.com";
newUser.LastLogon = DateTime.Now;

// Tell NHibernate that this object should be saved
session.Save(newUser);

// Commit all of the changes to the DB and close the ISession
transaction.Commit();
session.Close();

// Open another session to retrieve the just inserted user
session = factory.OpenSession();

User joeCool = (User)session.Load(typeof(User), "joe_cool");
```

```
// Set Joe Cool's Last Login property
joeCool.LastLogon = DateTime.Now;

// Flush the changes from the Session to the Database
session.Flush();

// Query all users
IList userList = session.CreateCriteria(typeof(User)).List();
foreach(User user in userList)
{
   System.Diagnostics.Debug.WriteLine(
     "{0} last logged in at {1}", user.Id, user.LastLogon);
}

// Query users who logged-on after a specified date
ICriteria criteria = session.CreateCriteria(typeof(User));
criteria.Add(Expression.Expression.Gt(
   "LastLogon", new DateTime(2005, 06, 14)));
IList recentUsers = criteria.List();

foreach(User user in recentUsers)
{
   System.Diagnostics.Debug.WriteLine(
     "{0} last logged in at {1}", user.Id, user.LastLogon);
}

// Tell NHibernate to close this Session
session.Close();
```

Ето през какви стъпки минахме:

1. Създадохме **`Configuration`** обект, отговорник за съответствията между .NET класовете и базата от данни. В случая той по указано име на асембли открива и обработва всички файлове, завършващи с **`.hbm.xml`**.
2. Създадохме сесия към базата от данни. **`ISession`** обектът представя връзка към базата от данни, а **`ITransaction`** е транзакция, управлявана от NHibernate.
3. Записахме обект в базата от данни и извлякохме набор от обекти. Видяхме колко прозрачно работят операциите по съхраняване и колко лесно е конструирането на заявки.
4. След приключване затворихме сесията, за да бъде освободена ADO.NET връзката, използвана от NHibernate.

Помощни инструменти за NHibernate

Голяма част от ръчната работа, която извършихме в примера, се поддава на автоматизация и можем да очакваме появяването на редица придружаващи инструменти за генерирането на схема на базата от данни, генериране на класове от mapping файлове и обновяване на схемата. Вече са разработени няколко независими средства, които да ни помагат при използването на NHibernate. Сред тях са CodeSmith шаблони за различни видове генерации (http://www.intesoft.net/nhibernate/), както и анализатор за HQL заявки (http://developer.berlios.de/projects/nqa/). Трябва да се отбележи и поддръжката на NHibernate в инструмента Codus (http://www.adapdev.com/codus/index.aspx).

Други възможности

NHibernate предлага още множество интересни възможности, които не успяхме да обхванем. Сред тях са: по-сложни съответствия от вида един-към-много, работа със сортирани и вложени колекции, настройки за повишаване на производителността и т.н. Можете да откриете допълнителна информация в документацията на NHibernate и в тази на по-зрелия му предшественик за Java – Hibernate.

Използвана литература

- Fredrik Normén, Persistence - http://fredrik.nsquared2.com/viewpost.aspx?PostID=209&showfeedback=true
- Scott Ambler, The Object-Relational Impedance Mismatch - http://www.agiledata.org/essays/impedanceMismatch.html
- Dino Esposito, DataSets vs. Collections - http://msdn.microsoft.com/msdnmag/issues/05/08/CuttingEdge/default.aspx
- NHibernate Quick Start Guide - http://wiki.nhibernate.org/display/NH/Quick+Start+Guide
- Tobins' NHibernate FAQ - http://www.tobinharris.com/nhibernatefaq.aspx

NAnt

NAnt е инструмент за автоматизиране на build процеса за едно приложение. Той предлага мощни възможности за управление на компилацията, конфигурацията и инсталацията на софтуерни компоненти. Разработчиците обикновено се стремят да автоматизират максимално тези процеси, тъй като ръчното им извършване носи излишни рискове от грешки. Чрез NAnt можем да опишем стъпките от процеса чрез удобен, XML базиран синтаксис и да го направим напълно повторяем. Проектът е опит да се изгради .NET аналог на Jakarta Ant build системата. Jakarta Ant е стандарт в автоматизирането на build процеса в Java средите, а NAnt се налага по

подобен начин в .NET света. Можете да изтеглите инструмента и документацията му от страницата на проекта http://nant.sourceforge.net.

Защо ни е нужен NAnt?

Защо ни е нужен инструмент за автоматизиране на компилирането? Нима Microsoft Visual Studio .NET не се справя достатъчно добре? Обикновено процесът на разработка на дадено приложение включва много повече задачи от компилацията. NAnt покрива всички стъпки по веригата от разработчика до потребителя. Скриптовете му могат да контролират издърпването на кода от система за контрол на версиите, компилирането, изграждането на MSI инсталационен пакет, пускането на автоматизирани тестове, копирането на файловете на определен за целта сървър и извършване на инсталацията. Всяка от тези операции е проста сама по себе си и обикновено не отнема много време. Това, обаче не е причина да не ги автоматизираме. Задачите се натрупват, губим много време и често правим грешки или забравяме нещо. Причиненото неудобство може да наруши периодичността на доставка на нови версии на продукта за тестване и употреба. Рядкото публикуване на нови версии носи рискове за проекта, като скрива евентуални проблеми и намалява шанса за навременна намеса.

Защо точно NAnt, а не някоя друга система или комбинация от любимия ни скриптов език с .BAT файлове? NAnt е много добре интегриран със съществуващата .NET инфраструктура и работи отлично с вградените инструменти. NAnt е мултиплатформен инструмент и поддържа както Microsoft .NET, така и Mono. Възможностите на NAnt могат да бъдат лесно разширявани. Той може да изпълнява скриптове и програми, написани на други езици, като по този начин играе ролята на лепило между различни вече съществуващи инструменти.

Основни функции

- Разделяне на проекта на цели (targets) и задачи (tasks). Поддържат се подпроекти и различни конфигурации.
- Описание на зависимостите между различните цели. Изпълнение на целите в реда на зависимостите.
- Удобна работа с файлове – лесно копиране, местене, архивиране, обновяване.
- Интеграция с популярните компилатори, системи за контрол на версиите и други инструменти.
- Различни нотификации и лог съобщения при евентуален неуспех на скрипта. Пълната поддръжка на XML базирани лог файлове и нотификация по електронна поща го правят лесен за интегриране в съществуващи системи.

- Разширяемост – добавяне на допълнителни функции и задачи, имплементирани на произволен .NET език.

Основни понятия

Всеки NAnt проект се състои от набор от цели, които се изпълняват, за да се постигне крайният резултат. Чрез указване на зависимости между целите се определя правилният им ред на изпълнение. Целите се описват чрез набор от команди, още известни като задачи. Да разгледаме един минимален скрипт, който дефинира прост NAnt проект с три цели:

default.build

```
<project default="compile">

  <target name="compile">
    <csc target="library" output="bin\MyWeb.dll">
      <sources>
        <include name="**\*.cs"/>
      </sources>
    </csc>
  </target>

  <target name="clean">
    <delete file="bin\MyWeb.dll"/>
  </target>

  <target name="rebuild" depends="clean,compile">
  </target>

</project>
```

Декларацията на **<project>** елемента описва целта, която се изпълнява по подразбиране. NAnt позволява изпълнението на произволна цел, чието име се подава от командния ред. Ако такова не е подадено, се изпълнява целта по подразбиране (с всичките й зависимости преди това).

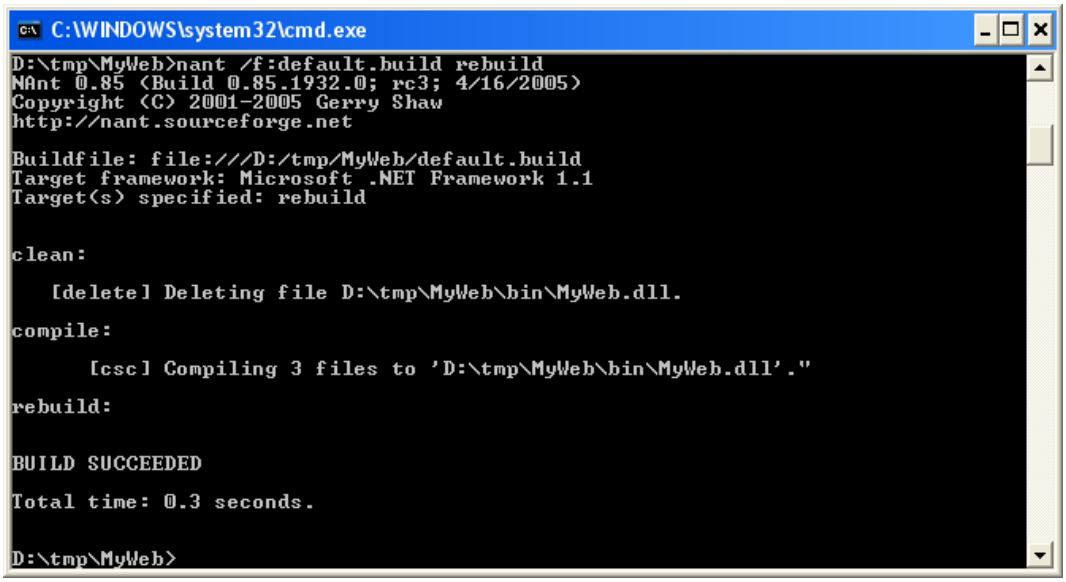
Основната цел в проекта ни е **compile**. Тя се грижи да извика C# компилатора и да компилира всички файлове с разширение ***.cs** в текущата директория и всички нейни поддиректории. NAnt следи датите на модификация на файловете и не компилира излишно, ако няма променени файлове след последната компилация.

Ако искаме да изчистим проекта от междинните файлове, генерирани от компилацията, можем да включим съответната цел **clean**, която да изтрие генерираното от компилатора асембли.

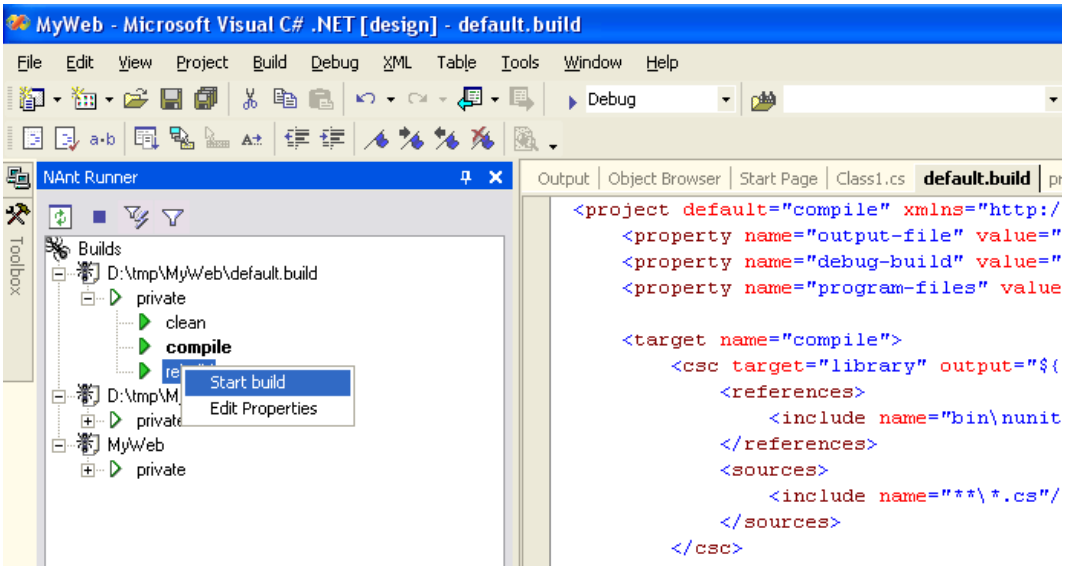
Пример за зависимостите между целите можем да видим в дефиницията на **rebuild** целта – тя просто предизвиква почистване, последвано от нова компилация.

Изпълнение на NAnt скриптове

NAnt скриптовете се изпълняват от конзолното приложение **nant.exe**. Най-важните параметри, които то приема, са името на скриптовия файл и името на целта за изпълнение:



Скриптовете се именуват с разширение **.build** и ако не е подадено име на файл текущата директория се претърсва за файлове с това разширение. В случай, че открие само един файл, то NAnt ще изпълни него. Ако има няколко, по подразбиране се изпълнява файла с име **default.build**.



Можем да изпълняваме скриптовете и от Microsoft Visual Studio.NET, ако сме инсталирали разширението NantRunner (вж. фигурата по-горе). От

неговия интерфейс можем да изберем с мишката скрипта и целта, която да изпълним.

NAntRunner може да бъде изтеглен от http://nantrunner.sourceforge.net. Друг удобен инструмент за изпълнение на скриптовете е NAntMenu (http://taschenorakel.de/mathias/nantmenu.en.html), който се интегрира в контекстните менюта на Windows Explorer.

Конфигурация на скриптовете

Скриптовете могат да бъдат конфигурирани чрез външни файлове, променливи от средата или параметри на командния ред. Основният елемент в конфигурацията и условното изпълнение са т. нар. свойства (properties) и функции. Свойствата се декларират с елемента **<property>**:

```
<property name="output-file" value="bin\MyWeb.dll" />
<property name="debug-build" value="true" overwrite="false"/>
<property name="program-files" value="${environment::get-folder-
path('ProgramFiles')}" />

<delete file="${output-file}"/>
<copy file="${output-file}" todir="${program-files}\NAntTest"/>
```

Използването на стойността на дадено свойство става с **${property-name}** синтаксиса. Можем да използваме вградени или външни функции при дефиницията на свойствата, както и да създаваме собствени стойности чрез вградената интерполация на низовете – **${program-files}\MyFolder**. Стойност на някое свойство може да бъде зададена от командния ред:

```
nant compile -D:debug-build=false
```

Ако искаме да предоставим стойност по подразбиране, трябва да дефинираме свойство със същото име в скрипта и да го маркираме с атрибута **overwrite="false"**. Това ще гарантира, че стойността, подадена на командния ред, няма да бъде презаписана с тази, която е дефинирана в скрипта. Така например можем да дефинираме цел за компилиране, която да компилира **debug** или **release** версия според подадените командни параметри:

```
<target name="compile">
  <csc target="library" output="${output-file}"
       debug="${debug-build}">
    <sources>
      <include name="**\*.cs"/>
    </sources>
  </csc>
</target>
```

Можем да декларираме общи свойства за проекта в отделен файл и да ги използваме навсякъде, където са ни необходими. Подходящи кандидати за отделяне са: базови имена на файлове, пътища към известни инструменти или версии. Така при нужда от корекции и разширения поддръжката на скриптовете ще бъде значително улеснена. Ето един пример:

config.build

```
<project name="MyWeb" default="all">
  <property name="version" value="1.3"/>
  <property name="staging-server" value="LocalTest"/>
</project>
```

Горният конфигурационен NAnt скрипт може да се използва от други NAnt скриптове чрез възможността за включване:

deploy.build

```
<project name="MyWeb" default="deploy">
  <target name="deploy">
    <include buildfile="config.build"/>
    <copy file="MyWeb_${version}.msi"
      todir="\\${staging-server}\MyWeb"/>
  </target>
</project>
```

Организация на сложни скриптове

Би било неефективно да опитваме да опишем целият build процес на голям проект в един скрипт. NAnt има вградени възможности за разделяне на скриптовете на компоненти и многократно използване на вече дефинирани стъпки от процеса. Технически, всеки проект може да бъде разделен на подпроекти, чиито build процес може да бъде описан в отделни скриптове. Общата практика е всеки подпроект да се помещава в отделна директория, която да съдържа скрипт, управляващ билда. Главният проект знае за съставните си части и се грижи да извика в правилния момент скриптовете на подпроектите. Нека разгледаме примерно ASP.NET приложение със следната структура на директориите:

- Кодът се намира в поддиректория "**Code**".
- Искаме да обфускираме (obfuscate), т. е. умишлено да направим нечетливи метаданните в асемблитата, за да защитим интелектуалната си собственост. Използваме за целта обфускатор (например Dotfuscator) и държим неговите файлове в поддиректория "**Obfuscation**".
- Искаме да предоставим MSI пакет. Проектът и допълнителните файлове държим в директория "**Installation**".

Всяка директория съдържа по един скрипт с име "**default.build**", който "знае" какво да направи за дадения подпроект. Можем да използваме **<nant>** задачата, за да извикаме тези скриптове от главния скрипт за проекта:

project.build

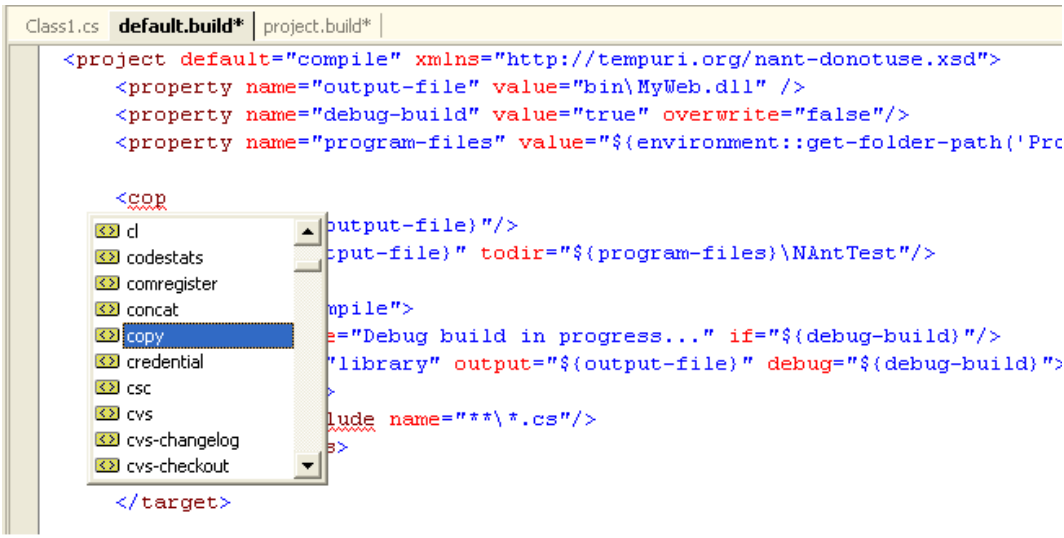
```
<project name="MyWeb" default="all">
  <target name="all">
    <nant buildfile="Code\default.build">
      <properties>
        <property name="debug-build" value="false"/>
      </properties>
    </nant>
    <nant buildfile="Obfuscation\default.build"/>
    <nant buildfile="Installation\default.build"/>
  </target>
</project>
```

Задачата за компилирането на кода приема като параметър флаг, дали да направи **debug** или **release** build. Можем да предадем този параметър с **<properties>** елемента на **<nant>**. Задачата може да бъде конфигурирана така, че дъщерният скрипт да наследи всички свойства на предшественика си, но това не е препоръчителна практика, тъй като може да причини трудно проследими проблеми в случай на съвпадащи имена на свойства.

Интеграция с Microsoft Visual Studio.NET

Един от най-често задаваните въпроси от разработчиците е "как да интегрирам този инструмент в моя проект, с моето копие на Visual Studio.NET". NAnt вече поддържа по-популярните компилатори за .NET езиците. Съществува допълнителен проект NAntContrib (http://nantcontrib.sourceforge.net) с голям набор от задачи, които не са част от базовата NAnt дистрибуция. Поддържат се разпространените системи за контрол на версиите: Visual Source Safe, CVS, Subversion, Perforce, ClearCase. Различни популярни сървъри като IIS на Microsoft също могат лесно да бъдат контролирани и конфигурирани. **<sql>** задачата позволява лесната работа с OLEDB съвместими бази данни, а задачите **<xmlpeek>** и **<xmlpoke>** позволяват четенето и манипулацията на XML данни.

Интеграцията с Visual Studio.NET също е на добро ниво. XML синтаксисът на NAnt проектите има дефинирана XSD схема, която можем да използваме, за да имаме IntelliSense подсказване докато пишем скриптовете:



Интеграция с NUnit

Препоръчително е автоматизираните тестове за проекта да се изпълняват при всеки build. Така се подсигуряваме, че приложението работи както очакваме и разбираме за евентуални дефекти възможно най-рано. NAnt поддържа най-популярната .NET библиотека за писане на unit тестове – NUnit. Освен задачата **<nunit2>**, е достъпна и оригиналната **<nunit>**, служеща за обратна съвместимост с по-старите версии NUnit 1.x.

Идеалното време за изпълнение на тестовете обикновено е след компилацията на асемблитата. На **<nunit2>** се подават като параметри имената на асемблитата с тестове, имената на тестовете (класовете, маркирани с **TestFixture** атрибута) и категориите от тестове, които трябва да се изпълнят. Ако не се подадат категории или имена на класове, се изпълняват всички тестове в асемблито:

test.build

```
<project default="test">
  <target name="test">
    <nunit2>
      <test>
        <assemblies>
          <include name="bin\MyWeb.dll"/>
        </assemblies>
        <categories>
          <include name="Smoke tests"/>
          <exclude name="Performance tests"/>
        </categories>
      </test>
      <formatter type="Plain"/>
    </nunit2>
```

```
  </target>
</project>
```

В примера сме изключили тестовете за производителност, тъй като обикновено те отнемат повече време.

Използвана литература

- NAnt Manual - http://nant.sourceforge.net/release/latest/help/
- NAntWiki - http://nant.sourceforge.net/wiki/index.php/HomePage
- Giuseppe Greco, Building Projects with NAnt - http://developer.agamura.com/technotes/building-projects-with-nant/index.html
- Nant-users mailing list - http://nant.sourceforge.net/mailinglists.html
- NAntContrib Manual - http://nantcontrib.sourceforge.net/release/latest/help/

Други помощни средства

В настоящата тема представихме библиотеки и инструменти, които могат да направят разработката на .NET приложения по-продуктивно и по-приятно занимание. Съществуват и много други помощни средства, които биха ни били полезни в определени ситуации. По-долу ще спомена и опишем съвсем накратко седем от тях. Отново няма да включваме комерсиални продукти, макар в някои важни области на .NET разработката (примерно при рефакторинг и обфускация), за момента да липсват безплатни алтернативи.

NDoc

NDoc (http://ndoc.sourceforge.net/) е генератор на документация от .NET асемблита и C# XML коментари. Разработени са и добавки (add-ins), поддържащи XML документационни коментари както за VB.NET, така и за управляван C++ код. Изходните формати за документацията включват MSDN-подобен HTML Help (`.chm`), Visual Studio .NET Help (HTML Help 2) и други. В случай, че се налага да документирате публичен API на библиотеки от класове, NDoc значително ще автоматизира и улесни работата ви.

GhostDoc

GhostDoc (http://www.roland-weigelt.de/ghostdoc/) е добавка към Visual Studio .NET за автоматично генериране на тези части от документационните коментари в C#, които могат да бъдат дедуцирани от името и типа на съответните методи, свойства, параметри и т.н. Ако задавате ясни, подробни и коректни имена на частите от кода, този инструмент ще ви спести усилия и време при създаването на качествени XML коментари.

Snippet Compiler

Snippet Compiler (http://www.sliver.com/dotnet/SnippetCompiler/) е приложение, с което да пишем, компилираме и изпълняваме C# и VB.NET код. За малки задачи може да бъде отличен заместник на Visual Studio .NET. Поддържа не малка част от възможностите, характерни за интегрираните среди за програмиране, като е далеч по-олекотен от големите комерсиални продукти. Ако нямате достъп до Visual Studio .NET или искате бързо да пробвате някакъв код, Snippet Compiler може да ви бъде отличен помощник.

ASP.NET Web Matrix

ASP.NET Web Matrix (http://www.asp.net/webmatrix/) е безплатен инструмент за разработка на ASP.NET приложения. Включва дизайнер за ASP.NET и HTML страници, интегриран е със SQL Server и MS Access, улеснява генерирането на интерфейсни компоненти, свързани с данни, идва със собствен уеб сървър и предлага още много други възможности. ASP.NET Web Matrix е отлична алтернатива на Visual Studio .NET при разработка на ASP.NET приложения.

Tree Surgeon

Tree Surgeon (http://confluence.public.thoughtworks.org/display/TREE/) е приложение, с което можем за броени секунди да подготвим пълно-функционална среда за разработка на нов .NET проект. След като зададем име на проекта, за нас ще бъде създадена структура от директории, включващи сорс файлове, помощни инструменти, референции и зависимости, оформени по последователен и лесен за интегриране и поддържане начин. Ако започвате разработка с Visual Studio .NET, NAnt и NUnit, то Tree Surgeon ще ви асистира в прилагането на множество добри практики за ефективната им съвместна употреба.

NDepend

NDepend (http://smacchia.chez.tiscali.fr/NDepend.html) е инструмент за вземане на метрики. Той анализира .NET асемблита и генерира метрики, свързани с качеството на дизайна: възможност за разширяемост, степен на преизползване, леснота на поддръжка. NDepend предоставя удобен преглед на топологията на приложението на ниво компоненти, типове и членове. Този инструмент може да ви помогне във формалната оценка на качество на кода и да ви подскаже кои части от приложението ви са най-уязвими.

CruiseControl.NET

CruiseControl.NET (http://ccnet.thoughtworks.com/) е инструмент за непрекъсната интеграция (continuous integration) по време на разработката. Той следи за промени в хранилището със сорс кода и щом открие такива,

автоматично извършва интеграционен build и валидира промените. Разработчиците могат да бъдат незабавно известявани по различни начини за текущия статус на системата. CruiseControl.NET е интегриран с много от популярните инструменти за контрол на версиите, build, тестване и метрики. Чрез практиката на непрекъсната интеграция, можете да избегнете много от проблемите, характерни за екипната разработка.

Портали за инструменти

За отправна точка при търсене на други помощни .NET средства можете да използвате портали като SharpToolbox (http://sharptoolbox.com/). SharpToolbox е събрал изключително богата колекция с инструменти, категоризирани по начин удобен за претърсване. Там можете да откриете подходящи решения за голяма част от предизвикателствата, с които ще се сблъскате. Ще можете и значително да обогатите списъка си от средства, които ще ви асистират в разработката на .NET приложения. С познанията ви за различните инструменти ще можете да решавате проблемите по ефективни начини и ще избегнете опасността, изказана в старата поговорка "Ако имате само чук в ръка, всичко останало Ви се вижда като пирон".

Упражнения

1. Какво е .NET Reflector? За какво служи?
2. С .NET Reflector декомпилирайте класа **`System.Collections.Hashtable`** и проверете какво е условието за преоразмеряване (resize) на хеш-таблица. Можете ли да изчислите ползвайки декомпилирания код колко пъти ще се извърши преоразмеряване при добавянето на 10 000 елемента последователно?
3. За какво служи инструмента FxCop? Кога трябва да се използва?
4. С помощта на FxCop анализирайте асемблитата от практическия проект от последната глава на настоящата книга. Намирате ли проблеми? (Ще се учудим много, ако няма никакви!)
5. За какво служи инструментът CodeSmith? Кога се ползва? Посочете няколко примера, в които е удачно да се ползва CodeSmith.
6. Напишете шаблони за CodeSmith, които по зададен connection string за достъп до SQL Server база данни генерира за всяка таблица от базата данни метод за извличане на всички нейни записи, която ги връща като ADO.NET **`DataTable`** обект.
7. Какво представляват unit тестовете в софтуерното инженерство? Кога се ползват и какво се постига чрез тях?
8. За какво служи инструментът NUnit? Кога се ползва? Как се създават unit тестове?

9. Напишете метод, който по даден текст намира най-често срещаната в него дума, а ако са няколко – първата от тях по азбучен ред. Напишете серия unit тестове, които проверяват дали методът работи коректно в различни случаи ситуации.

10. Какво представлява техниката "логинг" в софтуерното инженерство? Кога се използва?

11. За какво служи инструментът log4net? Кога се използва? Какви нива на логинг поддържа?

12. Напишете програма, която търси даден файл на твърдия диск. Добавете към нея логер, който запазва всички директории, които са намерени при търсенето.

13. Какво представлява концепцията "object relational mapping"? Кога се използва? Какво се печели от нея?

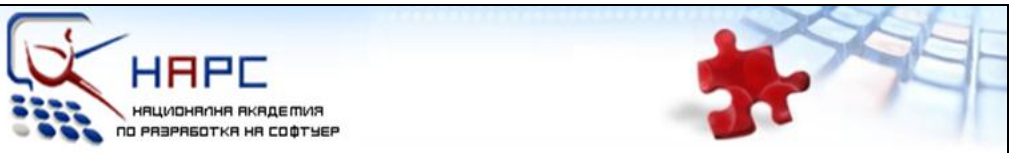
14. За какво служи инструментът NHibernate? Кога се използва?

15. Създайте база от данни в SQL Server за описание на дейността на магазин за хранителни стоки. Основните таблици в модела на данните са производители, продукти, клиенти и продажби. Дефинирайте C# класове, които съответстват на таблиците от базата данни. Дефинирайте XML mapping файлове, които задават съответствия между базата данни и C# класовете. Конфигурирайте Nhibernate за достъп до базата данни чрез дефинираните C# класове и съответствията им с базата данни. Реализирайте чрез стандартните класове от NHibernate основните операции с данните: извличане на списък, добавяне, промяна и изтриване на производители, продукти, клиенти и продажби.

16. Какво представляват средствата за построяване (build) на приложения? Какви действия включва построяването на един продукт?

17. За какво служи инструментът NAnt? Кога се използва и с каква цел?

18. Реализирайте NAnt скрипт, който извършва компилация и deployment на уеб услуга и уеб приложение върху отдалечен IIS сървър, достъпен по FTP.



НАЦИОНАЛНА АКАДЕМИЯ ПО РАЗРАБОТКА НА СОФТУЕР

Лекторите

» *Светлин Наков* е автор на десетки технически публикации и няколко книги, свързани с разработката на софтуер, заради което е търсен лектор и консултант.

Той е разработчик с дългогодишен опит, работил по разнообразни проекти, реализирани с различни технологии (.NET, Java, Oracle, PKI и др.) и преподавател по съвременни софтуерни технологии в СУ "Св. Климент Охридски".

През 2004 г. е носител на наградата "**Джон Атанасов**" на президента на България Георги Първанов.

Светлин Наков ръководи обучението по Java технологии в Академията.

» *Мартин Кулов* е софтуерен инженер и консултант с дългогодишен опит в изграждането на решения с платформите на Microsoft.

Мартин е опитен инструктор и сертифициран от Майкрософт разработчик по програмите MCSD, MCSD.NET, MCPD и MVP и международен лектор в световната организация на .NET потребителските групи INETA.

Мартин Кулов ръководи обучението по .NET технологии в Академията.

Академията

» **Национална академия по разработка на софтуер** (**НАРС**) е център за професионално обучение на софтуерни специалисти.

» **НАРС** провежда **БЕЗПЛАТНО** курсове по разработка на софтуер и съвременни софтуерни технологии в София и други градове.

» Предлагани специалности:

- **Въведение в програмирането (с езиците C# и Java)**
- **Core .NET Developer**
- **Core Java Developer**

» **Качествено обучение** с много **практически проекти** и индивидуално внимание за всеки.

» **Гарантирана работа!** Трудов договор при постъпване в Академията.

» БЕЗПЛАТНО!

Учите безплатно във въведителните курсове и по стипендии от работодателите в следващите нива.

http://academy.devbg.org

Глава 29. Практически проект

Автори

Ивайло Христов

Тодор Колев

Бранимир Ангелов

Ивайло Димов

Необходими знания

- Познания за архитектурата на .NET Framework
- Познания за езика C#
- Познаване на обектно-ориентираното програмиране в .NET Framework
- Познания за управление на изключенията в .NET Framework
- Познания за делегатите и събитията в .NET Framework
- Познания за масивите и колекциите в .NET Framework
- Познания за символните низове в .NET Framework
- Познания за вход и изход в .NET Framework
- Основни познания за работа с XML
- Познания за работа с релационни бази от данни и MS SQL Server
- Познания за достъп до данни с ADO.NET
- Познания за изграждане на графичен потребителски интерфейс с Windows Forms
- Познания за изграждане на уеб услугите с ASP.NET
- Познания и умения за изграждане на уеб приложения с ASP.NET

Съдържание

- Клиентски слой – Windows Forms GUI приложението
- Клиентски слой – ASP.NET уеб приложението
- Внедряване на системата

В тази тема …

В настоящата тема ще разгледаме как можем да приложим на практика технологиите, с които се запознахме в предходните теми. Ще си поставим за задача да разработим един сериозен практически проект – система за запознанства в Интернет с възможност за уеб и GUI достъп. Основното е, че това не е пример, какъвто бихте видели в повечето книги.

При реализацията на системата ще преминем през всички фази от разработката на софтуерни проекти: анализиране и дефиниране на изискванията, изготвяне на системна архитектура, проектиране на база от данни, имплементация и внедряване на системата.

При изготвяне на архитектурата ще разделим приложението на три слоя – база от данни (която ще реализираме с MS SQL Server 2000), бизнес слой (който ще реализираме като ASP.NET уеб услуга) и клиентски слой (който ще реализираме в две разновидности: ASP.NET уеб приложение и Windows Forms GUI приложение).

Система за запознанства в Интернет – визия

Основният мотив на потребителите да се регистрират в сайт за запознанства е вероятността да срещнат сродна душа. Потребителите вярват, че шансът да намерят подходящия човек е значително по-голям в сайтовете, в които има повече регистрирани потребители. В момента Интернет пространството се състои от множество малки сайтове (с малък брой потребители) и няколко сайта със значително количество регистрирани потребители. Тази ситуация мотивира създаването на единна система за запознанства, обединяваща потребителите на многото по-малки сайтове.

Една такава система не би трябвало да ограничава всеки, който иска да се включи да реализира по определен начин сайта си. Нужно е да се предостави възможност на всички клиентски уебсайтове да могат да използват услугите на системата, независимо от програмния език, на който са реализирани и платформата, върху която се изпълняват.

Основно системата трябва да предоставя възможност за:

- регистрация на нови потребители;
- търсене на потребители по различни критерии;
- организиране на част от потребителите на системата в категории;
- обмяна на съобщения между потребителите;
- запазване на всички разменени съобщения;
- статистика за потребителите.

За още по-лесно включване в системата трябва да има и реализирано клиентско приложение, състоящо се от уеб приложение и GUI (десктоп) приложение. Уеб приложението трябва да предоставя основната функционалност, а GUI приложението да предоставя улеснен достъп до най-използваните функции на системата.

Някои системи за обмяна на съобщения съхраняват всички разменени съобщения локално на компютъра на потребителя и съответно при включване в системата от друг компютър, човекът няма достъп до диалозите, които е провел. Системата трябва да решава и този проблем.

Какво е функционална спецификация?

Функционалната спецификация е документ, който описва в детайли функционалните изисквания към системата (Software Requirements Specification - SRS). Написването на функционална спецификация подпомага изграждане на обща визия за софтуерния продукт между клиентите и разработчиците. Тя помага и за по-детайлно изясняване на това какво може и би трябвало да прави продуктът.

След продължителни разговори и уточняване на изискванията, обсъждане на различни варианти и предложения достигнахме до функционална спецификация.

Функционални възможности на системата за запознанства

Да се реализира система за запознанства по Интернет. Системата трябва да се състои от ASP.NET-базиран уебсайт за запознанства и Windows Forms-базирано клиентско приложение. Трябва да се реализира функционалност за регистрация на потребители, търсене на потребители и обмяна на съобщения между потребителите.

Функционални възможности на ASP.NET уеб приложението

1. Управление на потребителите и техните профили:

1.1. Идентификация на потребител (login, logout):

- Посетителите на сайта (без автентикация) имат достъп само до формата за регистрация.
- Идентифицираните (влезлите успешно в системата) потребители имат достъп до всички функционални възможности без възможностите за администрация на системата.

2. Потребителски профил

2.1. За всеки потребител се пази профил, който съдържа следната информация:

- потребителско име (позволени са всички потребителски имена съставени от букви, цифри и символите "-", ".", "_", без запазеното потребителско име "Administrator");
- парола;
- e-mail адрес (необходимо е въвеждането на валиден e-mail адрес);
- име и фамилия, град, пол, рождена дата;
- снимка – не е задължителна.

2.2. Системата позволява на всеки потребител да редактира собствения си профил.

3. Регистрация на нов потребител

3.1. При регистрация на нов потребител той въвежда цялата информация за профила си.

3.2. Въведеният e-mail адрес се удостоверява чрез изпращане на произволно генерирана парола (на потребителя не се предоставя възможността сам да избере парола).

3.3. За защита от автоматична регистрация при регистрацията на потребител динамично се генерира картинка, съдържаща трудна за четене последователност от няколко цифри, които потребителят трябва да въведе (и така доказва, че не е бот).

4. Следене на активност

Системата следи активността на всеки потребител, като записва датата и часа на последното му действие.

5. Търсене на потребител по различни критерии

5.1. Търсенето се извършва по следните критерии: град, пол, възраст. Предоставя се възможност за търсене само по един от критериите или по произволна комбинация от тях.

5.2. При всяко търсене резултатите се сортират по брой разглеждания на профила.

5.3. Ако резултатите от търсенето са повече от 10, се дава възможност за страниране и навигация между страниците (визуализират се по 10 резултата на страница).

5.4. Потребителят може да разглежда профила на всеки потребител получен като резултат от търсенето и по желание да добавя този потребител в "списък с приятели".

6. Поддръжка на "списък с приятели" за всеки потребител

6.1. За всеки потребител се поддържа списък от любими потребители, наречен "списък с приятели". Списъкът е организиран в категории само на едно ниво.

6.2. При създаване на нов профил в "списък с приятели" се създава автоматично категорията "Нови приятели".

6.3. Потребителят може да редактира категориите от своя "списък с приятели". Позволени са следните действия:

- добавяне на категории;
- изтриване на категории, при което:
  - всички записи в категорията се изтриват;
  - не се позволява изтриването на категорията "нови контакти";
- промяна на името на категория.

6.4. Потребителят може да редактира своя "списък с приятели" чрез следните действия:

- добавяне и изтриване на потребители от "списък с приятели";
- добавянето на "приятели" става само в някоя от вече съществуващите категории (единствено чрез функционалността за търсене на потребител);
- изтриването на потребител става с потвърждение;
- преместване на потребители от една категория в друга.

7. Разглеждане на профили

Всеки потребител може да разглежда профилите на потребителите от своя "списък с приятели", както и профилите на потребителите, които е намерил чрез търсене.

8. Обмяна на съобщения между потребителите

8.1. Всеки идентифициран потребител може да влиза в режим на диалог с всеки потребител от своя "списък с приятели" (може и с няколко едновременно).

8.2. В режим на диалог потребителят може да изпраща на другия съобщения и едновременно с това да наблюдава списъка с последните 50 съобщения, обменени между двамата. Този списък се обновява на всеки 5 секунди.

8.3. Не е задължително ако един потребител е в режим на диалог с друг, другият също да е в режим на диалог с първия.

8.4. Изпращането на съобщения е позволено дори ако получателят не е влязъл в системата в дадения момент.

8.5. За всяко съобщение освен текста му се пази дата и час на изпращане.

8.6. Ако се получи съобщение от потребител, който не е от "списъка с приятели", този потребител се добавя автоматично в категорията "нови приятели".

8.7. При получаване на ново съобщение, изпратено от друг потребител, ако има отворен диалог между двамата, съобщението се визуализира в него, а в противен случай в "списъка с приятели" за съответния потребител се появява специална индикация за чакащи съобщения.

9. Администрация на системата

9.1. Административната подсистема позволява достъп само на администратора на системата чрез потребителско име и парола.

9.2. За идентификация се ползва запазеното потребителско име "Administrator".

9.3. Администраторът може да изтрива потребители от системата – при изтриване на потребител се изтриват всички негови данни (профил, съобщения и контакти).

9.4. Администраторът може да редактира профилите на потребителите.

9.5. Администраторът може да извлича всички разменени съобщения между произволни два потребителя.

10. Статистика

10.1. За всеки потребител се пази броя разглеждания на профила му до момента.

10.2. Броят регистрирани потребители е нужно да се показва на началната страница.

Функционални възможности на Windows Forms клиентското приложение

1. Идентификация на потребител

Системата поддържа идентификация на потребителите (login, logout) и позволява достъп само след успешна автентикация.

2. Търсене на потребител по различни критерии

Търсенето като функционалност е еднакво с търсенето в уеб-приложението, но връща само първите 200 резултата.

3. Управление на "списък с приятели"

Потребителят има достъп до своя "списък с приятели" и може да извършва с него всички действия, които могат да се извършват от уеб приложението.

4. Система за обмен на съобщения

Потребителите могат да влизат в режим на диалог с потребители от своя "списък с приятели" и да обменят съобщения с тях, както при уеб-приложението.

Нефункционални изисквания към системата за запознанства по Интернет

1. Изисквания за ASP.NET уеб приложението

При реализация на ASP.NET уеб приложението за случаите, в които се изисква автентикация на потребителите трябва да се ползва вградената в ASP.NET технология "Forms Authentication".

2. Изисквания за данните

Всички данни на системата трябва да се съхраняват в базата от данни. Не се допуска използване на други механизми за съхранение на данни, като например файловата система.

3. Изисквания за сигурност на данните

Всички пароли трябва да се предават във вид, в който не могат да бъдат директно прочетени (кодирани).

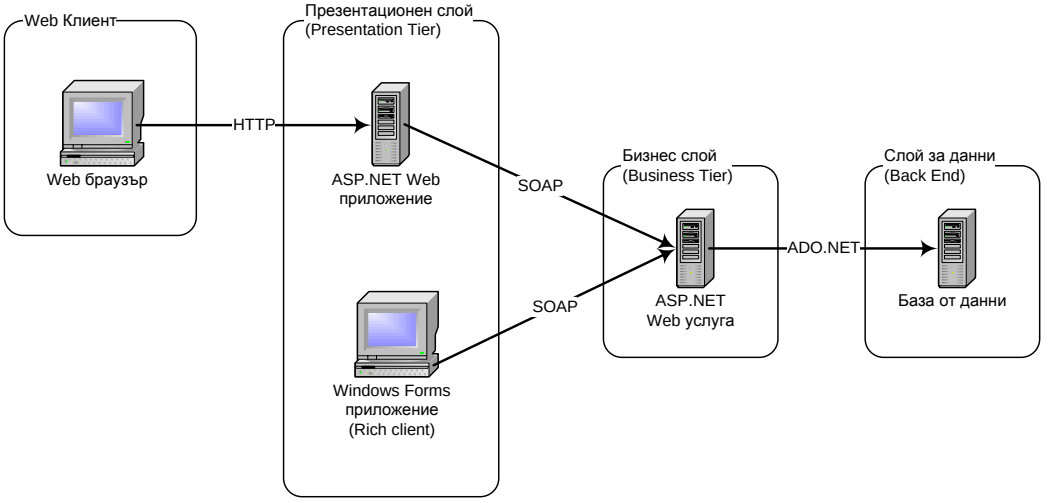
4. Хардуерни изисквания

Системата трябва да работи на стандартен компютър с 1GHz CPU, 256 RAM, 1GB дисково пространство, върху Windows ОС. Уеб приложението трябва да поддържа Internet Explorer 5.0/6.0/7.0 и Mozilla Firefox 1.x.

Архитектура на системата

Ще реализираме системата, използвайки класическа Enterprise архитектура, базирана на многослоен модел за разпределени приложения (моделът на .NET Enterprise приложенията). Този модел се състои от:

1. Презентационен слой (Front End) – потребителски интерфейс (Windows Forms и ASP.NET уеб приложения).
2. Бизнес слой (Business Tier) – бизнес логиката на системата (ASP.NET уеб услуга).
3. Слой за данните (Back End) – данните на системата (база данни SQL Server).



В използвания трислоен модел комуникацията между отделните слоеве се извършва по правилото, че всеки слой комуникира само със съседния си, както е указано на схемата, т.е.:

1. Презентационният слой управлява взаимодействието с потребителя и изпраща заявки към бизнес слоя. Забранена е директна комуникация с базата данни, както и с други компоненти на презентационния слой.
2. Бизнес слоят реализира работните процеси и операциите над данните. Той предоставя съвкупност от бизнес операции над данните в системата и си комуникира с базата от данни. Бизнес слоят комуникира с базата данни и презентационния слой.
3. Слоят за данни се реализира от релационна база от данни, в която данните се съхраняват в таблици с връзки между тях. Слоят за данни комуникира само с бизнес слоя.

Ще използваме трислойната архитектура заради нейната гъвкавост и разпределеност. Гъвкавостта на тази архитектура се изразява в това, че всеки слой е максимално самостоятелен. Това дава възможност всеки слой лесно и сравнително независимо от останалите да бъде разширяван и дори подменян. Очаква се освен уеб и GUI (десктоп) да има и други клиенти към системата като Flash приложения и Java аплети. Именно мощта на трислойната архитектура и използването на SOAP базирана уеб услуга позволява това да бъде лесно реализирано в бъдеще без промяна в бизнес слоя и слоя за данни.

Разпределеността на трислойната архитектура от своя страна позволява всеки слой да бъде разположен физически на отделен компютър и дори да се изгради клъстер от множество компютри за даден слой. По този начин производителността на системата може на практика да бъде разширявана неограничено и да поема все по-големи натоварвания. Така се покриват изискванията на очакваното голямо натоварване на система от такъв тип.

Имплементация на системата

При начинаещите програмисти се наблюдава тенденцията да започват работата от потребителския интерфейс и изграждайки потребителски интерфейс да имплементират нужната им функционалност. В общия случай това е погрешен подход. Ще започнем изграждането на системата от базата данни. След това ще изградим уеб услугата. След като имаме напълно изградена уеб услугата може да се работи паралелно по Windows Forms и ASP.NET клиентите.

Слой за данни

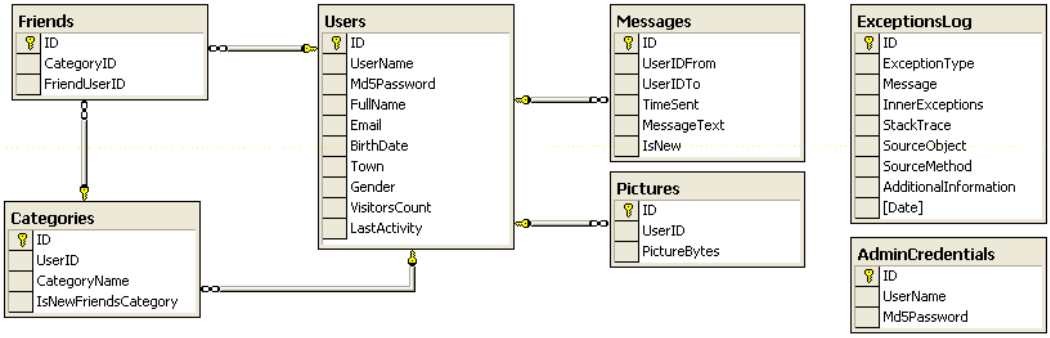
Слоят за данни е мястото, където се съхраняват данните, необходими на системата да функционира. Това може да бъде текстов файл като XML (Extensible Markup Language) файл, CSV (Comma Separated Value) файл и др. При по-големи приложения и системи, които изискват надеждно съхраняване на голямо количество информация е необходимо използването на бази от данни. Те осигуряват по-удобен и оптимизиран начин за

съхранение на данните и по-бърз достъп до тях. За нуждите на системата е избрана MS SQL Server база данни. Причините за това са няколко:

- SQL Server е сървър за управление на релационни бази от данни на Microsoft, който има много добра поддръжка в .NET Framework и е най-честият избор на разработчиците при създаването на .NET решения.
- Очаква се системата за запознанства да има много потребители и следователно висока натовареност. Необходимо е мощно средство, което да съхранява голямо количество информация и да осигурява нейната бърза обработка. Безспорно такова е MS SQL Server и за това е избран при изграждане архитектурата на системата.

Проектиране на базата данни

Базата данни се състои от седем таблици: **`Users`**, **`AdminCredentials`**, **`Messages`**, **`Categories`**, **`Friends`**, **`Pictures`**, **`ExceptionsLog`**. Ето как изглежда моделът на данните:



Ето краткото описание на всяка от тях:

- **`Users`** – съдържа информация за потребителите на системата. В нея се съхраняват както потребителското име и MD5 хеша на паролата му, така и личните данни на базата, на които се осъществява търсенето на потребители в системата.
- **`AdminCredentials`** - потребителите на системата, които са администратори са отделени в отделна таблица, защото за тях не е необходима допълнителна лична информация, както за останалите потребители.
- **`Messages`** – съдържа съобщенията, които се разменят между потребителите като се пази от кого и за кого е съобщението, а също така дали е ново.
- **`Categories`** – съдържа категориите с приятели на потребителите на системата. Интересно поле в тази таблица е **`IsNewFriendCategory`**. То е флаг, който показва дали категорията е нормална или е такава, в която се записват всички нови приятели на потребителя.

- **Friends** – съдържа приятелите на потребителите в системата. Таблицата се състои от три полета: идентификатор (id) на записа в таблицата, id на категорията, към която принадлежи този приятел и потребителското id на приятеля.
- **Pictures** – съдържа снимките на потребителите като масив от байтове.
- **ExceptionsLog** – представлява log с всички изключения в уеб услугата.

Имплементиране на логиката на ниво база данни

Основен подход при имплементацията на логиката на ниво база от данни е, че достъпът до данните става само през запазени процедури. Така се постига уеднаквяване и се обособява още един слой на ниво база от данни, който повишава нивото на абстракция и улеснява добавянето на нова функционалност или променянето на стара такава в клиентското приложение (уеб услугата).

Всяка запазена процедура изпълнява една проста заявка. Затова може да се разграничат най-общо четири типа запазени процедури: за **Select**, **Insert**, **Update** и **Delete**. Почти винаги дадена заявка се изпълнява върху една таблица, като целта е цялата бизнес логика да се извършва от уеб услугата, а запазените процедури да реализират само основните операции. Така запазените процедури по никакъв начин не знаят каква специфична логика ще се реализира в уеб услугата. Същевременно те предоставят една добра абстракция, чрез която уеб услугата може лесно да извършва базовите операции с данни, които са ѝ необходими.

Ето и кода на една от многото запазени процедури:

```
ALTER PROCEDURE spU_ChangeCategoryName
(
   @ID int ,
   @CategoryName varchar(50)
)
AS

UPDATE [Categories]
SET
   [Categories].[CategoryName] = @CategoryName
WHERE
   [Categories].[ID] = @ID

RETURN @@ROWCOUNT
```

Тя приема два параметъра:

- **@ID** - id на категорията, на която искаме да променим името;
- **@CategoryName** – новото име на категорията.

След като завърши **Update** заявката върху таблицата, запазената процедура връща променливата **@@ROWCOUNT**, която показва броя редове, които са се променили в резултат на заявката. Ако той е нула, това показва, че не съществува категория с такова **id** и по този начин засичаме, че запазената процедура е извикана с грешни аргументи. Този механизъм за валидация на аргументите се използва в почти всички запазени процедури от тип **Update** и **Delete**. При запазените процедури от тип **Insert** се връща **SCOPE_IDENTITY()**, което представлява идентификатора (ID) на новодобавения запис. Това се използва, за да не се прави допълнителна заявка за извличането му.

Останалите запазени процедури работят на подобен принцип като горе илюстрираната процедура и читателят може сам да ги разгледа и разучи.

Съхранена процедура за търсене на потребители

Търсенето на потребители в системата се реализира от съхранената процедура **spS_SearchUsers**, която е малко по-сложна от останалите:

```
ALTER PROCEDURE spS_SearchUsers
(
   @PageIndex int = 0,
   @PageSize int = 10,
   @Town varchar(50) = NULL,
   @Gender char(1) = NULL,
   @AgeFrom int = NULL,
   @AgeTo int = NULL,
   @GetAllResultsCount bit = 0
)
AS

SET NOCOUNT ON

DECLARE
   @TownFilter varchar(50),
   @AgeFilter varchar(100),
   @GenderFilter varchar(50),
   @Concatinator varchar(5)

SET @Concatinator = '';
SET @TownFilter = '';
SET @GenderFilter = '';
SET @AgeFilter = '';

IF( @Town IS NOT NULL )
BEGIN
   SET @TownFilter = ' Town = @Town ';
   SET @Concatinator = ' AND ';
END
```

```
IF( @Gender IS NOT NULL )
BEGIN
   SET @GenderFilter = @Concatinator + ' Gender = @Gender ';
   SET @Concatinator = ' AND ';
END

IF( (@AgeFrom IS NOT NULL) AND (@AgeTo IS NOT NULL) AND @AgeTo >
@AgeFrom )
BEGIN
   SET @AgeFilter =  @Concatinator + ' DATEDIFF(Year, BirthDate,
     getdate()) BETWEEN @AgeFrom AND @AgeTo ';
END

DECLARE @Sql nvarchar(512)
DECLARE @Where varchar(255)

SET @Where = ''

IF ( @TownFilter <> '' OR @GenderFilter <> '' OR @AgeFilter <>
   '' ) BEGIN
   SET @Where = ' WHERE ' + @TownFilter + @GenderFilter +
     @AgeFilter + ' '
END

IF ( @GetAllResultsCount = 1 ) BEGIN
   SET @Sql  = 'SELECT COUNT(ID) FROM Users ' + @Where
END ELSE BEGIN
   DECLARE @WhereConcatinator varchar(10)

   IF (@Where <> '') BEGIN
     SET @WhereConcatinator = ' AND ';
   END ELSE BEGIN
     SET @WhereConcatinator = ' WHERE ';
   END

   SET @Sql  = 'SELECT TOP ' + CAST ( @PageSize as varchar ) +
     ' * FROM  Users '
     + @Where + @WhereConcatinator + ' ID NOT IN ( SELECT TOP '
     + CAST ( (@PageSize * @PageIndex) as varchar )
     + ' ID FROM Users ' + @Where
     + ' ORDER BY VisitorsCount DESC, ID ) ORDER BY
     VisitorsCount DESC, ID'
END

DECLARE @ParametersList nvarchar(255)
SET @ParametersList = '@Town varchar(50), @Gender char(1),
   @AgeFrom int, @AgeTo int'

EXECUTE sp_executesql @Sql, @ParametersList, @Town=@Town,
@Gender=@Gender, @AgeFrom=@AgeFrom, @AgeTo=@AgeTo;
```

```
RETURN @@ROWCOUNT
```

Освен търсенето на потребители, тази процедура има за задача и страниране на резултатите. То трябва да стане на възможно най-ниско ниво в архитектурата, за да бъде избегната загубата на системни ресурси, породена от пренасянето на голям брой данни между отделните слоеве.

Функционалността на тази запазена процедура е реализирана посредством съставянето на динамична заявка. За да бъдат избегнати атаки от типа "SQL injection" тази заявка бива изпълнявана с параметри чрез физическо конкатениране на параметризирани стойности в текста на заявката. Това става със SQL командата **EXECUTE**, която изпълнява вградената в MS SQL Server запазена процедура **sp_executesql**. Тази процедура приема низ на заявката, която да изпълни, низ с описание на параметрите на тази заявка и самите параметри, с които тя да бъде изпълнена.

Бизнес слой – ASP.NET уеб услугата

Бизнес слоят на системата реализира основната функционалност на системата, т. нар. "бизнес логика", т.е. работните процеси и правилата за обработка на данни, които сървърът извършва, за да обслужи отделните клиенти.

Дизайн съображения, свързани с уеб услугата

Дизайнът на уеб услугата е направен с цел да осигури максимална съвместимост с клиенти реализирани чрез различни езици, използващи различни платформи. Пример за такива платформи, които с голяма вероятност биха се използвали в един реален сайт за запознанства са Java аплетите и Macromedia Flash приложенията.

За да се покрият тези изисквания при проектирането на системата са взети редица решения, свързани с избягването на технологии и практики, които са специфични за .NET Framework или не се подържат от по-слаби клиенти като Macromedia Flash. Тези решения са следните:

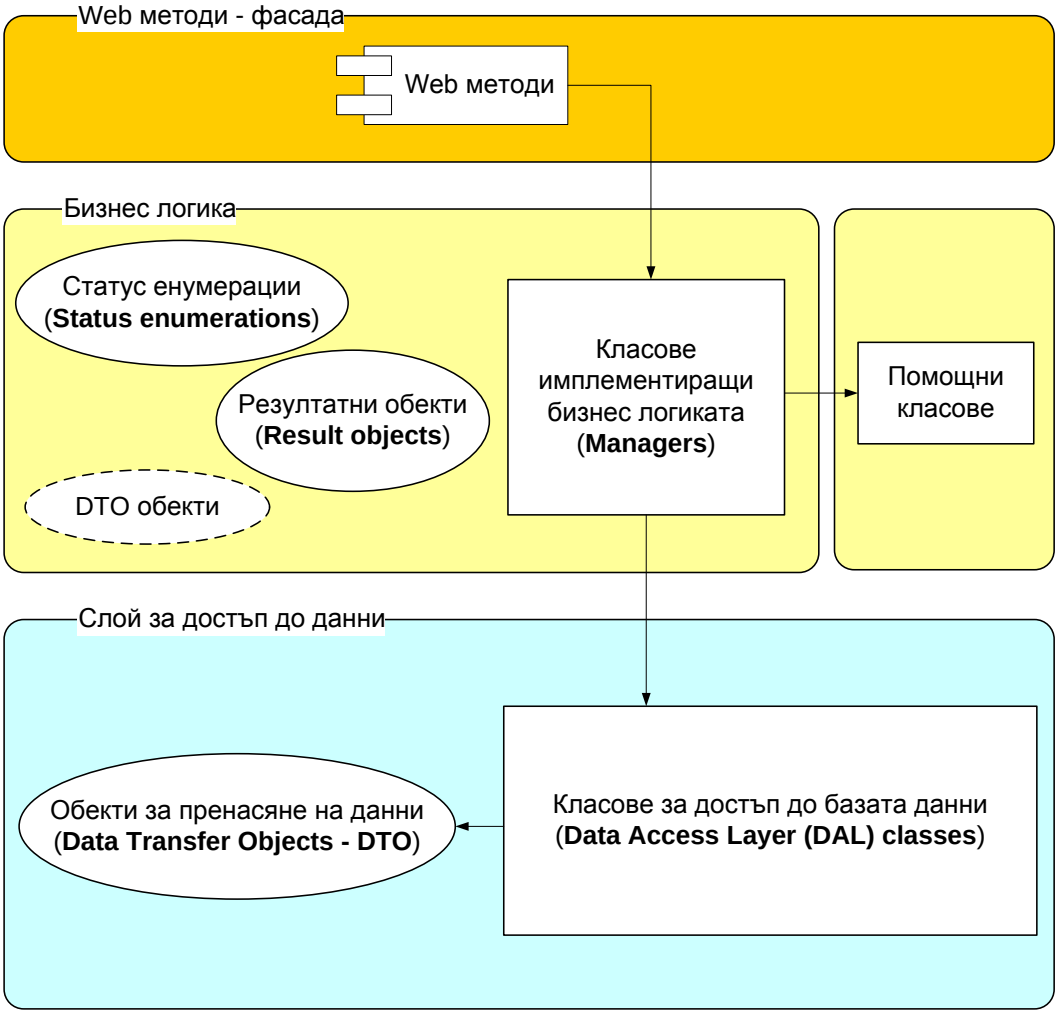
- За връщане на резултата се използват лесните за обработка Data Transport Object (DTO) обекти вместо **DataSet** обекти.
- Резултатът върнат от всеки метод съдържа информация за статуса на изпълнение на съответния метод, т.е. дали той е бил изпълнен успешно или е възникнало изключение. Това замества предизвикването на изключения (**SoapException**) от метода, тъй като не всички платформи биха ги обработили правилно.
- Реализиран е собствен механизъм за управление на сесиите. Идентификаторът на сесията се предава като обикновен аргумент на уеб методите, които го изискват. Това се налага поради факта, че ASP.NET сесията е базирана на бисквитки (Cookies), а те не се

подържат от всички платформи, които се предвижда да бъдат клиенти на уеб услугата.

- Поради големия обем данни, съдържащи се в снимките, се налага те да бъдат предавани като поток. Техниките за реализиране са специфични за .NET Framework или са част от Web Service Enhancements, които не се подържат от всички нужни платформи. Това налага реализирането на `.aspx` страница, която да връща снимките като поток и да бъде извиквана като обикновен ресурс през HTTP заявка.

Архитектура на уеб услугата

Ето как изглежда диаграмата, която визуализира отделните слоеве на уеб услугата и взаимодействието между тях:



както се вижда от диаграмата, уеб услугата е изградена от три подслоя:

- слой за достъп до данни;

- бизнес логика;
- фасадни уеб методи.

Съществуват и помощни (utility) класове, които предоставят функционалност за изпращане на e-mail, хеширане и т.н. Тази архитектура е нарушена единствено при реализирането на функционалността за получаване снимката на потребител. В този случай бизнес логиката е енкапсулирана в класа на ASP.NET страница, която се извиква директно, вместо в **`Manager`** клас, който да бъде викан през уеб метод.

Дизайн решения в уеб услугата

Достъпът до данните се извършва от слоя за достъп до данни. Той се състои от няколко класа, всеки от които е отговорен за достъпа до един тип обекти (една таблица) от базата данни. Тези класове предоставят методи, всеки от които служи за викане на съхранена процедура (stored procedure) от базата данни. Някои от тези методи приемат като аргумент инстанция на транзакция в базата данни. Ако бъде подадена транзакция, то запазената процедура ще бъде изпълнена в контекста на тази транзакция. Създаването и управлението на транзакциите става от методите в слоя за бизнес логика. Ако не бъде подадена транзакция, чиято връзка към базата данни да бъде използвана, методите за достъп до данни сами създават връзка.

Методите за достъп до данни, селектиращи записи от базата данни връщат като резултат т.нар. обекти за пренос на данни. Тези обекти имат за цел единствено да служат за типизиран контейнер на данни. Резултатните обекти от своя страна представляват структури от данни, съдържащи обекти за пренос и информация за статуса на резултата като стойност от съответния изброен тип (enumeration). Този статус показва, че методът е бил изпълнен успешно или указва типа изключение, което е възникнало.

При изготвяне дизайна на уеб услугата е обърнато внимание на управлението на изключенията и грешките. При възникване на изключение в някой от методите от слоя за достъп до данни или помощните класове това изключение бива обвито в специфично изключение, съответстващо на абстракцията на съответния метод (**`DalException`**, **`SendMailException`** и т.н.), след което бива хвърлено отново. Всеки от тези методи се грижи при възникване на изключение да освободи заеманите от него критични ресурси като връзки към базата данни и UI обекти.

Методите на **`Manager`** класовете от своя страна след хващане на изключения ги записват в лога на изключенията и връщат съответния резултат, представляващ стойност от изброени тип за статус, на клиента на уеб услугата.

Имплементация на ASP.NET уеб услугата

Да разгледаме някои по-важни моменти от имплементацията на уеб услугата, която реализира работната логика (бизнес логиката) на системата за

запознанства в Интернет. Основната задача на уеб услугата е да прави връзка с базата данни и да изпълнява обработка на данните.

Класове за достъп до данните (DAL)

Достъпът до данните се реализира чрез Data Access Layer (DAL) – съвкупност от класове, които реализират логиката за достъп до данните, намиращи се в SQL Server, чрез ADO.NET и извикване на съхранени процедури.

При имплементирането на слоя за данни е създаден базов клас (**BaseDAL.cs**), който да бъде наследен от останалите класове за достъп до данни. Негова основна роля е да енкапсулира помощните методи, използвани в работата на наследяващите го класове. Такива са например методите за създаване на връзка към базата данни и обект за команда към базата данни (**SqlCommand**). Ето кода на втория метод:

```
private static SqlCommand GetSqlCommand(
   string aStoredProcedureName, SqlTransaction aTransaction,
   SqlConnection aSqlConnection)
{
   SqlCommand sqlCommand = null;
   try
   {
      if (aTransaction != null)
      {
         sqlCommand = new SqlCommand(aStoredProcedureName,
            aTransaction.Connection);
      }
      else
      {
         sqlCommand = new SqlCommand(aStoredProcedureName,
            aSqlConnection);
      }
      sqlCommand.CommandType = CommandType.StoredProcedure;
      if (sqlCommand.Connection.State == ConnectionState.Closed)
      {
         sqlCommand.Connection.Open();
      }
      if (aTransaction != null)
      {
         sqlCommand.Transaction = aTransaction;
      }
      return sqlCommand;
   }
   catch (Exception ex)
   {
      throw new DalException(ex.Message, ex);
   }
}
```

Пример за клас от DAL слоя

Да разгледаме класовете от DAL слоя на системата. За всяко entity таблица от базата данни е реализиран по един съответен DAL клас в слоя за достъп до данните.

Ето и един от типичните методи за достъп до данни, извличащ информацията за потребител по неговия идентификатор (ID) в таблицата от базата данни:

```
public static UserDTO SelectUser(SqlInt32 aID)
{
   SqlCommand sqlCommand = null;
   SqlDataReader reader = null;
   try
   {
      sqlCommand = GetSqlCommand("spS_User");
      sqlCommand.Parameters.Add("@ID", SqlDbType.Int)
         .Value = aID;
      reader = sqlCommand.ExecuteReader();
      if(reader.HasRows)
      {
         reader.Read();
         return GetUserFromReader(reader);
      }
      else
      {
         return new UserDTO();
      }
   }
   catch(Exception ex)
   {
      throw new DalException(ex.Message, ex);
   }
   finally
   {
      if(reader != null)
      {
         reader.Close();
      }
      if(sqlCommand != null)
      {
         sqlCommand.Connection.Close();
      }
   }
}
```

За пренос на данните се използват Data Transfer Object (DTO) обекти. Те представляват прости структури от данни, съответстващи на полетата от базата данни.

Основни проблеми при реализацията на бизнес слоя

Имплементирането на методите от слоя за бизнес логика е свързано с няколко основни задачи. За справянето с тях е необходим единен подход, който да бъде използван консистентно. Можем да идентифицираме следните проблемни ситуации:

- проверка на валидността на сесията;
- осигуряване правилното изпълнението на съответната бизнес функционалност чрез проверка на входните данни и подържане на сесия;
- същинско изпълнение на функционалността чрез викане на методи от слоя за достъп до данни (DAL);
- проверка за правилното изпълнение на методите от слоя за достъп до данни;
- записване в лога на възникналите изключения;
- връщане на резултат или тип на възникналото изключение, ако има такова.

Пример за бизнес метод от работната логика на системата

Ето кода на един типичен метод, имплементиращ бизнес логиката за смяна на името на категория с приятели:

```
public AddRenameCategoryStatus RenameCategory(int aCategoryID,
   string aNewName, string aSessionID )
{
   SqlTransaction sqlTransaction = null;
   SqlConnection sqlConnection = null;
   try
   {
      sqlConnection = BaseDAL.GetSqlConnection();
      sqlConnection.Open();
      sqlTransaction = sqlConnection.BeginTransaction(
         IsolationLevel.Serializable);
      int categoryOwnerUserID =
         CategoriesDAL.GetCategoryOwnerUserID(aCategoryID);
      bool isSessionValid = SessionManager.Instance
         .IsSessionForUser(aSessionID, categoryOwnerUserID);
      if(!isSessionValid)
      {
         sqlTransaction.Rollback();
         return AddRenameCategoryStatus.InvalidSession;
      }
      bool categoryNameExist = CategoriesDAL
         .CheckCategoryNameExist(categoryOwnerUserID, aNewName);
      if(categoryNameExist)
      {
         sqlTransaction.Rollback();
```

```
            return AddRenameCategoryStatus.CategoryNameExist;
        }
        int rowsAffected=CategoriesDAL.ChangeName(aCategoryID,
            aNewName, null);
        if (rowsAffected != 1)
        {
            sqlTransaction.Rollback();
            return AddRenameCategoryStatus.OperationNotPerformed;
        }
        sqlTransaction.Commit();
        return AddRenameCategoryStatus.Success;
    }
    catch(Exception ex)
    {
        if(sqlTransaction != null)
        {
            sqlTransaction.Rollback();
        }
        ExceptionHandler.HandleException(ex);
        return AddRenameCategoryStatus.InternalServerError;
    }
    finally
    {
        sqlConnection.Close();
    }
}
```

Управление на сесиите

Разработването на собствен механизъм за подържане на потребителска сесия изисква реализирането на клас, управляващ сесиите – **SessionManager**. Този клас използва **Hashtable** обекти, в които съхранява информацията за сесиите (време до изтичане на сесията и потребител, притежаващ сесията).

Класът **SessionManager** е реализиран като singleton (клас, за който съществува единствена инстанция), защото на практика на приложението е нужен един единствен **SessionManager**. Инстанцията на този клас се съхранява в негова статична член-променлива, като по този начин се осигурява единствеността на инстанцията.

Тъй като се предвижда методите на инстанцията на класа **SessionManager** да бъдат викани от множество нишки едновременно, поради голямо натоварване на сайта, съществува опасност данните в неговите **Hashtable** член-променливи да бъдат неправилно прочетени или записани. За да се избегне това всяка операция с **Hashtable** обектите бива поставена в **lock** блок. Този блок заключва статичната член-променлива **mSyncRoot**, като докато тя е заключена нито една друга нишка не може да я заключи или да изпълни блок с операция с **Hashtable** обекти. Така всички нишки ще изчакат заключилата **mSyncRoot** нишка да излезе от **lock** блока.

Ето кода на един от методите на класа **SessionManager**, който използва **lock** блок, за да се подсигури, че никой няма да създаде нова сесия, докато се проверява дали генерираният идентификатор на сесия е уникален:

```
public string StartUserSession(int aUserID)
{
   string sessionID = "";
   lock (mSyncRoot)
   {
      do
      {
         sessionID = GenerateSessionID();
      } while(mSessionTimeouts.ContainsKey(sessionID));

      mUserSessions.Add(sessionID, aUserID);
      mSessionTimeouts.Add(sessionID, mSessionTimeout);
   }

   return sessionID;
}
```

Извличане на снимките на потребителите

Друг интересен момент от имплементацията на уеб услугата е извличането и връщането на снимките на потребителите като поток. Предаването на данни като поток означава, че не се зареждат всички данни едновременно в паметта и след това да се предават като голям блок памет, а вместо това те биват накъсвани на много малки парчета, които биват предавани последователно. Не се предава следващият пакет, докато не бъде получен предишният и паметта, заемана от него, не бъде освободена. Така се получава нещо като поток от малки по обем пакети от данни, което намалява значително заеманата памет.

Предаването на снимките като поток съдържа в себе си две подзадачи – четене на снимките от базата данни като поток и пращането им към клиента като поток. Поточното четенето от базата данни е реализирано с метода **GetBytes()** на класа **SqlDataReader**. Това е методът от слоя за достъп до данни, който реализира поточното четене на снимките, който приема за параметър и потока, в който пише:

```
public static bool GetUserPictureAsStream(SqlInt32 aUserID,
   Stream aOutputStream)
{
   SqlCommand sqlCommand = null;
   SqlDataReader reader = null;

   try
   {
      sqlCommand = BaseDAL.GetSqlCommand("spS_PictureBytes");
```

```
      sqlCommand.Parameters.Add("@UserID", SqlDbType.Int)
         .Value = aUserID;
      reader = sqlCommand.ExecuteReader();

      if(reader.HasRows)
      {
         reader.Read();

         Int64 bytesRead;
         Int64 dataIndex = 0;
         byte[] buffer;

         buffer = new byte[PICTURE_BUFFER_LENGTH];
         do
         {
            bytesRead = reader.GetBytes(0, dataIndex, buffer, 0,
               PICTURE_BUFFER_LENGTH);
            dataIndex += bytesRead;

            aOutputStream.Write(buffer, 0, (int) bytesRead);
         }
         while (bytesRead > 0);

         return true;
      }
      else
      {
         return false;
      }
   }
   catch(Exception ex)
   {
      throw new DalException(ex.Message, ex);
   }
   finally
   {
      if(reader != null)
      {
         reader.Close();
      }
      if(sqlCommand != null)
      {
         sqlCommand.Connection.Close();
      }
   }
}
```

За да бъдат предадени данните като поток през HTTP от **.aspx** страницата е нужно да се укаже тя да не бъде буферирана и ако има буферирано съдържание, то да бъде изтрито. Тъй като предаваме снимки е нужно да

се укаже, че типа на предаваното съдържание е JPEG. Всичко това става със следните редове:

```
Response.BufferOutput = false;
Response.Clear();
Response.ContentType = "image/jpeg";
```

Методът **GetUserPictureAsStream()** бива извикан в страницата **get_picture.aspx**, като за стойност на параметъра **aOutputStream** бива предаден изходният поток на страницата:

```
bool getUserPictureResult = PicturesDAL.GetUserPictureAsStream(
  userID, Response.OutputStream);
```

Клиентски слой – Windows Forms GUI приложение

Основната задача на Windows Forms приложението е да предостави удобен и лесен за използване интерфейс за работа с най-често използваните функционални възможности на сайта за запознанства.

Дизайн съображения, свързани с Windows Forms клиента

Преди да започнем да разглеждаме как е имплементирана системата трябва да обърнем внимание на някои дизайн решения.

Решения за достъп до уеб услугата

Проблемът при изпълняване на метод от уеб услугата е, че блокира нишката, от която се изпълнява. Извикването на методите на уеб услугата от нишката на потребителския интерфейс би попречило на обновяването на потребителския интерфейс и потребителят би видял бял екран.

Когато добавяме уеб услугата в нашия проект, Visual Studio предоставя синхронни и асинхронни методи в автоматично генерирания прокси клас. Ако уеб услугата предоставя метод **OurMethod()**, освен синхронния метод **OurMethod()** в прокси класа ще има и два други метода **BeginOurMethod()** и **EndOurMethod()**.

Извиквайки **BeginOurMethod()**, трябва да подадем като параметър метода, който искаме да бъде изпълнен след като приключи изпълнението на му. Извиквайки **BeginOurMethod()** в нишка от пула за нишки ще се изпълни методът **OurMethod()** и след като приключи ще изпълни метода, подаден като параметър. За да получим резултата от метода **OurMethod()** е нужно да изпълним **EndOurMethod()**. Вече получили данните от уеб услугата, нормалното нещо, което бихме искали да направим, е да използваме тези данни за обновяването на някоя контрола от потребителския интерфейс. Това обаче е неправилно, защото все още се намираме в нишката от пула за нишки. Обновявайки потребителския интерфейс от нишка, различна от нишката, която го управлява би могло да предизвика неприятни последствия. Намирайки се в нишката от пула за нишки, можем

да изпълним метод в нишката на потребителския интерфейс чрез **`Control.Invoke()`**.

Това налага следния дизайн – за всяка една форма, която изпълнява методи от уеб услугата ще създадем клас, който се грижи за извикването на методите от уеб услугата.

Автентикация и управление на сесиите

Потребителите на Windows Forms приложението се автентикират, предоставяйки потребителско име и парола. При успешна автентикация уеб услугата ни предоставя низ, представляващ сесия. Всяка операция, извършвана от приложението, използва тази сесия, за да доказва самоличността на потребителя. При изтичане на сесията потребителя е помолен да въведе отново паролата си. При правилно въвеждане приложението продължава работата си.

Управление на изключенията и грешките

При всяко десктоп приложение най-важното е изключенията да бъдат прихванати, обработени и да бъде показано подходящо съобщение на потребителя. Ако приложението не може да се възстанови от грешката е нужно да се отбележи това с подходящо съобщение и да се излезе от приложението "културно", опитвайки се да се запази всякаква информация, която не е била запазена. В приложението, което разработваме, почти всичко се пази от уеб услугата и следователно трябва само да се погрижим да изведем подходящо съобщение. В приложението всички необработени изключения ще обработваме от един глобален **`try catch`** блок. Това обаче не е достатъчно. За да прихванем всички изключения е нужно да обработим събитията **`System.Windows.Forms.Application.ThreadException`** и **`AppDomain.CurrentDomain.UnhandledException`**.

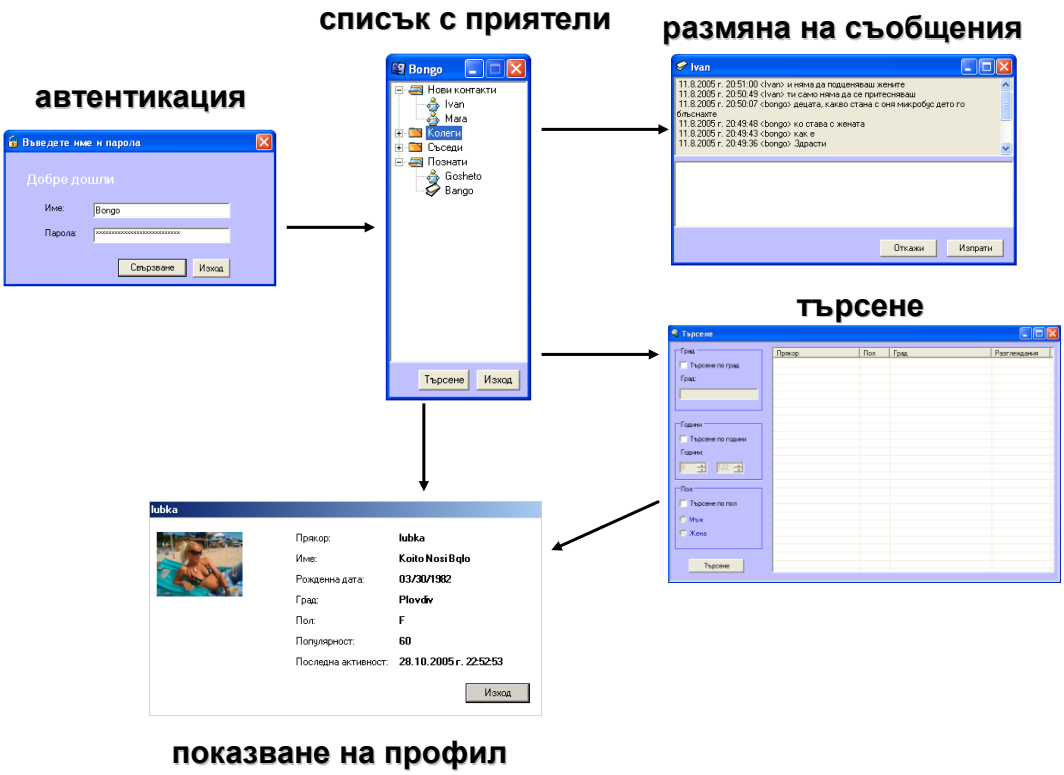
Имплементация на Windows Forms клиента

GUI приложението трябва да предоставя достъп само до регистрирани потребители и съответно всеки потребител, който иска да влезе в системата трябва да докаже самоличността си. Съответно първото нещо, което трябва да се покаже, е форма, в която се изисква въвеждане на потребителско име и парола.

Основната форма, която ще създадем ще съдържа списъка с категориите и приятелите във всяка категория. От тази форма ще можем да отворим друга форма, позволяваща размяната на съобщения между потребителите. От основната форма ще добавим възможност за отваряне на форма, предоставяща функционалността за търсене. За всеки един потребител, намерен като резултат от търсенето, ще предоставим възможността за извличане и показване на подробна информация. Това ще реализираме, чрез добавянето на една форма съдържаща информацията за потребителя, както и умалена снимка на потребителя. Би било добре възмож-

ността за разглеждане на профила да е достъпна и за всеки от приятелите в списъка с приятели.

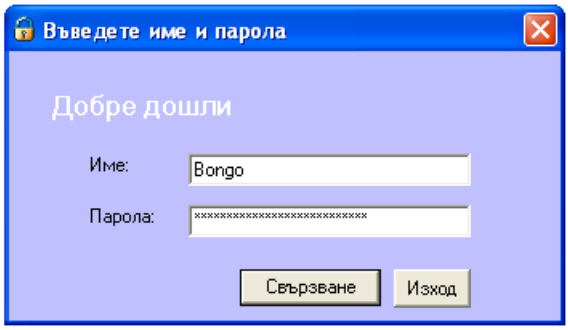
Ето една диаграма, която показва екраните на приложението и преходите между тях (screen flow diagram):



Форма за автентикация

Нека първо създадем формата за автентикация на потребителите – **`LogInForm`**. Има два случая, в които можем да използваме тази форма. Първият случай е когато потребител влиза в приложението и желае да докаже самоличността си. При втория случай, потребителят вече е работил с приложението и сесията му е изтекла и съответно се нуждае да получи нова сесия. Разликите между двата случая са, че в първия ще накарваме потребителя да въведе потребителското си име и парола, а във втория само да въведе паролата си. За целта ще имплементираме два конструктора на този клас. Единият конструктор ще приема потребителското име като параметър и ще го използва за автентикация с въведената от потребителя парола.

Ето как изглежда формата за автентикация:



Потребителят трябва или да се автентикира успешно или да натисне бутона "Изход", като и в двата случая трябва да се затвори формата, но в различните ситуации трябва да предприемем различни действия. Нужно е след затваряне на формата да знаем какъв е бил резултатът от автентикацията. За целта ще използваме изброения тип **`DialogResult`**. Нужно е да отбележим, че след затварянето на формата, тя не е унищожена, което ни позволява да използваме **`DialogResult`** след затварянето ѝ, но ни задължава след това да извикаме метода **`Dispose()`**, за да може формата и всички използвани от нея ресурси да бъдат освободени.

При натискане на бутона "Свързване", ще направим неизползваеми бутона "Свързване" и текстовите полета за въвеждане на потребителско име и парола. Това ще покаже на потребителя, че трябва да изчака докато приложението свърши с това действие.

След това трябва да изчислим MD5 хеш стойността на паролата. Ще използваме същите методи, които бяха използвани в уеб услугата. За целта ще създадем отделен помощен клас **`Utils`**, в който и за в бъдеще ще добавяме методи, полезни на приложението, но имащи функция, различна от функциите на приложението.

```
LogInForm logInForm = new LogInForm();
DialogResult result = logInForm.ShowDialog();
logInForm.Dispose();
if(result == DialogResult.OK)
{ ... }
else
{ ... }
```

Извикване на методи от уеб услугата

Сега вече сме готови за извикването на метода от уеб услугата за автентикация на потребителя. Уеб услугата ни предоставя метод **`AuthenticateUser()`**, който връща резултат, показващ дали данните, въведени от потребителя са правилни. Ако се върнем малко назад и погледнем дизайн съображенията, то ще видим, че ни е нужен и един помощен клас, в който ще са всички извиквания към уеб услугата от дадена форма. Този клас ще се казва **`LogInProxy`** и в него ще имплементираме следния метод:

```
public void AuthenticateUser(string aUserName, string
   aPasswordHash)
{
   mProxy.BeginAuthenticateUser(aUserName, aPasswordHash,
      new AsyncCallback(AuthenticateUserCallback), null);
}
```

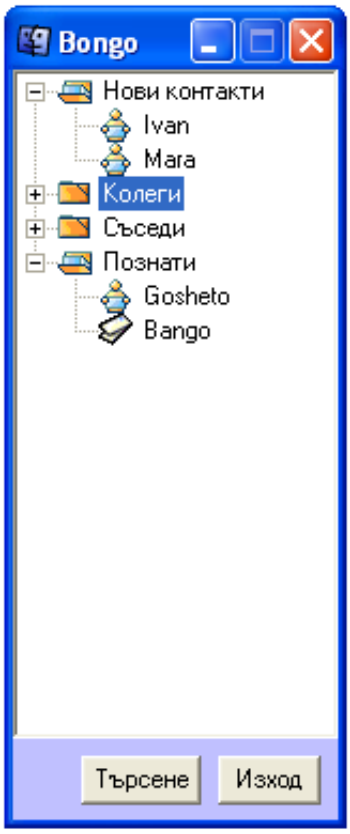
В този метод **mProxy** е инстанция на класа, който е автоматично генериран при добавяне на уеб услугата. Уеб услугата предоставя метод **AuthenticateUser()**, а прокси класът предоставя асинхронния метод **BeginAuthenticateUser**. На този метод трябва да подадем като параметри не само нужните на **AuthenticateUser** параметрите, но и два допълнителни параметъра. Първият от допълнителните параметри е делегат, указващ функцията, която ще се изпълни след като приключи изпълнението на метода от уеб услугата, а втория параметър ще разгледаме подробно малко по-надолу.

```
private void AuthenticateUserCallback(IAsyncResult aAsyncResult)
{
   try
   {
      AuthenticateUserResult result = mProxy.EndAuthenticateUser(
         aAsyncResult );
      mTargetForm.Invoke( new AuthenticateUserCompleteCallback
         (mTargetForm.AuthenticateUserComplete), new object[]
         {result});
   }
   catch(System.Net.WebException)
   {
      mTargetForm.Invoke(new WSNotFoundCallback
          (mTargetForm.WSNotFound));
   }
   catch(System.Web.Services.Protocols.SoapException)
   {
      Core.WebServiceException();
   }
}
```

Ако възникне изключение при достъпа или изпълнението на метода от уеб услугата, то това изключение ще се получи в десктоп приложението при извикването на метода **EndAuthenticateUser()**. За това е нужно да оградим извикването му в **try catch** блок. Ако възникне изключение от тип **WebException**, значи имаме проблем с достъпа до уеб услугата и съответно трябва да предприемем нужното действие. В случая ще извикаме метод от **LogInForm**, който показва подходящо съобщение. При възникване на **SoapException** изключение означава, че има изключение в уеб услугата. При нормално протичане на метода сме готови да изпълним метода **AuthenticateUserComplete()** от **LogInForm**. Както отбелязахме

преди, изпълнението на операции по интерфейса е нужно да се прави само от нишката на потребителския интерфейс. За това изпълняваме метода посредством **`Invoke()`**.

Изграждане на основната форма



Сега нека изградим основната форма, която ще визуализира приятелите, разделени в категории и ще предостави основната функционалност на приложението. Ще визуализираме приятелите, използвайки **`TreeView`** контрола. Отделянето на извикванията на методите към уеб услугата в отделен клас е аналогично на това, разгледано преди малко. За това сега няма да се спираме подробно.

Една от основните функции на приложение от този тип е да може лесно да показва на потребителите, ако някой им е изпратил съобщение. Ние ще имплементираме показването чрез смяна на иконката. Така, когато потребителят погледне, ще може бързо и лесно да забележи кой му е изпратил съобщение.

Обновяване на списъка с приятели

Понеже списъка с приятели може да бъде променян, не е допустимо да го заредим само в началото и да остане така до края. Имайки предвид, че потребител, който не е в списъка с приятелите ни, може да ни изпрати съобщение и тогава той трябва да се появи в списъка, то би трябвало да обновяваме постоянно този списък. Обновяване на 10 секунди би било в рамките на нормалното, но списъка с приятелите може да значително голям и съответно изтеглянето му на всеки 10 секунди би затруднило системата. За да се избегне това ще изтегляме съдържанието само на категорията "нови приятели" – това е единственото място, където може да се добави нов приятел без наше знание.

Когато се получи ново съобщение е нужно потребителят да бъде уведомен. Един добър начин да направим това е да сменим иконката в **`TreeView`** контролата на приятелите, които са изпратили съобщение. Съответно би трябвало постоянно да проверяваме дали има нови съобщения от всички в списъка с приятели и при нужда да сменяме иконката.

Използване на Tag полето

При построяването на дървото ще използваме **`Tag`** полето на **`TreeNode`** класа за съхраняване на допълнителна информация. За всяка категория ще добавяме **`CategoryDTO`** обект, представящ съответната категория, а за

приятелите ще добавяме **FriendDTO**. Тази информация ще ни е нужна при бъдеща работа с елементите на **TreeView** контролата и предимно в случаите, в които ще се налага да определим съответния елемент какви данни от базата представлява.

Създаване на менюта

Приложението трябва да предоставя различни функционални възможности за категориите и приятелите. За целта ще създадем две различни контекстни менюта. За работата с категориите в приложението ще използваме контекстното меню **ContextMenuCategory**, а за работа с приятелите - менюто **ContextMenuFriend**. При натискане на десен бутон на мишката ще се показва едно от двете менюта – ако курсорът е върху приятел ще се показва **ContextMenuFriend**, а ако е върху категория - ще се показва менюто **ContextMenuCategory**. Това ще реализираме със следния код:

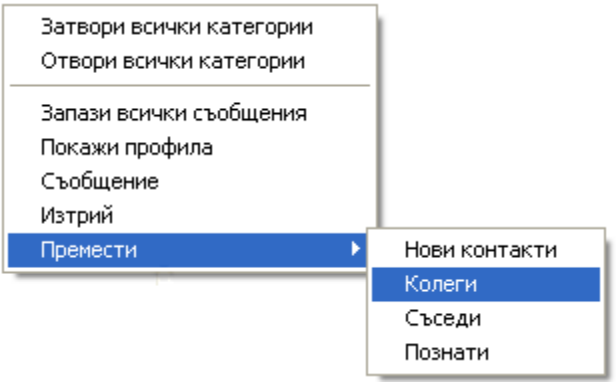
```
private void TreeViewFriends_MouseDown(object sender,
   System.Windows.Forms.MouseEventArgs e)
{
   if (e.Button == MouseButtons.Right)
   {
      TreeView senderControl = ( TreeView ) sender;
      TreeNode mouseNode = senderControl.GetNodeAt( e.X, e.Y );
      if( mouseNode != null )
      {
         senderControl.SelectedNode = mouseNode;
         if( mouseNode.Tag is FriendDTO )
         {
            ContextMenuFriend.Show( senderControl,
               new Point( e.X, e.Y ));
         }
         if( mouseNode.Tag is CategoryDTO )
         {
            ContextMenuCategory.Show( senderControl,
               new Point( e.X, e.Y ));
         }
      }
   }
}
```

За да можем да определим кое меню да покажем ще използваме, обекта в **Tag** полето, като проверяваме дали е **CategoryDTO** или **FriendDTO**.

Създаване на динамично меню

Сега ще имплементираме функционалността за преместване на приятел от една категория в друга. Трябва ни начин, позволяващ на потребителя да избере в коя категория желае да премести съответния приятел. Възможно е да реализираме тази функционалност чрез отварянето на нова форма, позволяваща избора на една от съществуващите категории, но това би

направило потребителския ни интерфейс по-нелогичен и по – труден за работа. За това ще се спрем на варианта да добавим едно подменю, което съдържа имената на всички категории:



Не е възможно създаването на това меню да стане в началото както създадохме предните две, защото потребителят може да изтрива категории или да добавя нови. Съответно при всяко показване на менюто **ContextMenuFriend** динамично ще създаваме ново подменю.

Бихме желали, когато се натисне някой от елементите на подменюто да се извика метод, който да знае коя категория от менюто сме избрали. Този метод ще получи като параметър обекта, който е предизвикал това събитие, а именно **MenuItem** обект. Това събитие ще трябва да премести потребителя в категорията, която сме избрали, а за да направи това би трябвало да има информация коя е категорията. Би било идеално при създаване на динамичното меню да запазим в **Tag** полето на **MenuItem** информация за категорията и при извикване на метода, обработващ даденото събитие да имаме нужната ни информация. За съжаление **MenuItem** класа няма **Tag** поле. Можем да се справим с този проблем като създадем клас наследник на класа **MenuItem**, който да има **Tag** поле:

```
public class CustomMenuItem : MenuItem
{
   private Object mTag;
   public Object Tag
   {
      get
      {
         return mTag;
      }
      set
      {
         mTag = value;
      }
   }
}
```

Ще имплементираме създаването на динамичното меню по следния начин:

```
private void CreateDynamicMenu()
{
   FriendsMoveMenu.MenuItems.Clear();
   ContextMenuFriend.MenuItems.Remove(FriendsMoveMenu);
   FriendsMoveMenu = new MenuItem();
   FriendsMoveMenu.Text = MOVE_MENU_TEXT;
   foreach(TreeNode categoryNode in TreeViewFriends.Nodes)
   {
      CustomMenuItem newMenuItem = new CustomMenuItem();
      CategoryDTO category = (CategoryDTO) categoryNode.Tag;
      newMenuItem.Text = category.Name;
      newMenuItem.Tag = categoryNode.Index;
      newMenuItem.Click += new EventHandler(this.MoveFriend);
      FriendsMoveMenu.MenuItems.Add(newMenuItem);
   }
   ContextMenuFriend.MenuItems.Add(FriendsMoveMenu);
}
```

Обхождат се всички категории и за всяка категория се създава **`CustomMenuItem`**. В **`Tag`** полето запазваме индекса на категорията в дървото. Това ни е нужно, защото при преместването на един приятел в друга категория трябва не само да извикаме метода на уеб услугата, а и да обновим потребителския интерфейс. Запазвайки индекса на елемента от дървото, можем да обновим потребителския интерфейс и имаме възможност да извлечем информация за категорията от **`Tag`** полето на елемента от дървото:

```
private void MoveFriend(object sender, System.EventArgs e)
{
   CustomMenuItem menuItem = (CustomMenuItem) sender;
   int categoryTreeIndex = (int) menuItem.Tag;
   TreeNode categoryNode =
      TreeViewFriends.Nodes[categoryTreeIndex];
   CategoryDTO category = (CategoryDTO) categoryNode.Tag;
   TreeNode friendNode = TreeViewFriends.SelectedNode;
   FriendDTO friend = (FriendDTO) friendNode.Tag;
   mFormProxy.MoveFriend( friend.ID, category.ID,
      Core.SessionID, friendNode, categoryTreeIndex );
}
```

Предаване на данните

Специално внимание ще обърнем на имплементирането на функционалността за изтриване на категория. Нужно е да извикаме метода **`DeleteCategory()`** на уеб услугата и е нужно да изтрием елемента от дървото. Избирайки от менюто командата за изтриване не ни гарантира, че тази категория ще бъде изтрита. Възможно е категорията да не може да бъде изтрита и в такъв случай не би трябвало да я премахваме от

дървото. Съответно премахването на елемента, представляващ тази категория от дървото, трябва да стане едва след като методът на уеб услугата върне резултат показващ, че изтриването е успешно. За да направим това на нас ние е нужно да предаваме данни, с които да разполагаме в метода **`DeleteCategoryComplete()`**. Това може да стане, като предадем желаната информация през **`AsyncState`** параметъра на метода **`BeginDeleteCategory()`**:

```
public void DeleteCategory(int aCategoryID,
     int aCategoryTreeIndex, string aSessionID )
{
   mProxy.BeginDeleteCategory(aCategoryID,aSessionID, new
     AsyncCallback(DeleteCategoryComplete), aCategoryTreeIndex);
}
```

Впоследствие можем да получим тази стойност по следния начин:

```
private void DeleteCategoryComplete( IAsyncResult aAsyncResult )
{
   try
   {
      int aCategoryTreeIndex = (int) aAsyncResult.AsyncState;
      MethodStatus status=mProxy.EndDeleteCategory(aAsyncResult);
      mTargetForm.Invoke(new DeleteCategoryCompleteCallback
         (mTargetForm.DeleteCategoryComplete), new object[]
         {status, aCategoryTreeIndex});
   }
   catch(System.Net.WebException)
   {
      mTargetForm.ConnectionLost();
   }
   catch(System.Web.Services.Protocols.SoapException)
   {
      Core.WebServiceException();
   }
}
```

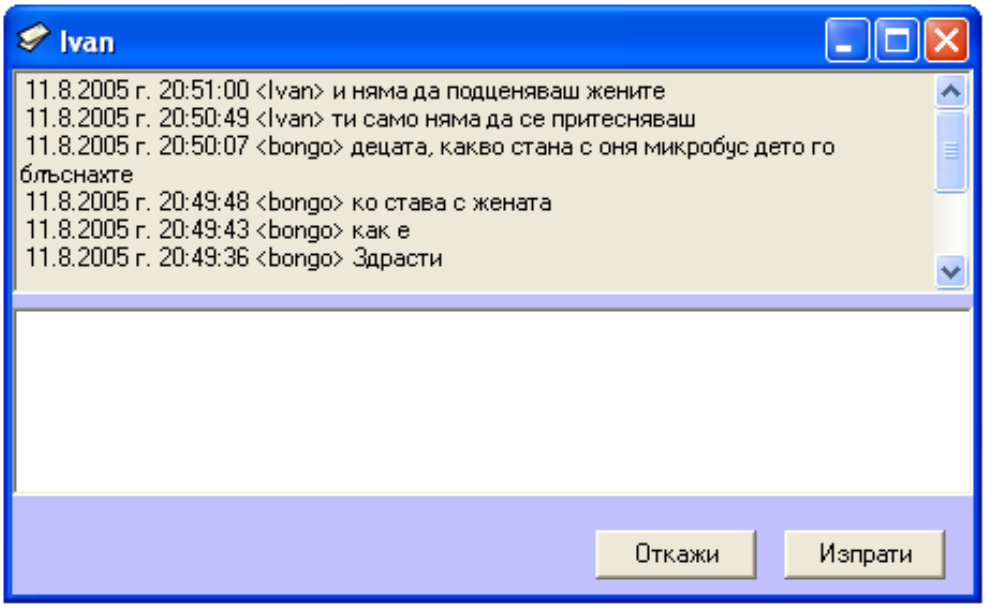
Сега ще създадем метода **`DeleteCategoryComplete()`**. Това е методът, който ще бъде извикан, за да обнови потребителския интерфейс:

```
public void DeleteCategoryComplete(MethodStatus aStatus,int
   aCategoryTreeIndex)
{
   Core.ProceedMethodStatus(aStatus);
   if(aStatus == MethodStatus.Success)
   {
      TreeViewFriends.Nodes[aCategoryTreeIndex].Remove();
   }
}
```

Ще използваме метода **`ProceedMethodStatus()`** за обработване на статуса, върнат от уеб услугата. Този метод има за цел да покаже подходящо съобщение за грешка, ако изпълнението не е било успешно или да извика метода **`UpdateLastActivity()`**, ако всичко е преминало успешно. Ако и изпълнението е минало успешно, искаме да премахнем от дървото категорията.

Форма за изпращане на съобщения

Нека сега изградим формата за изпращане на съобщения:



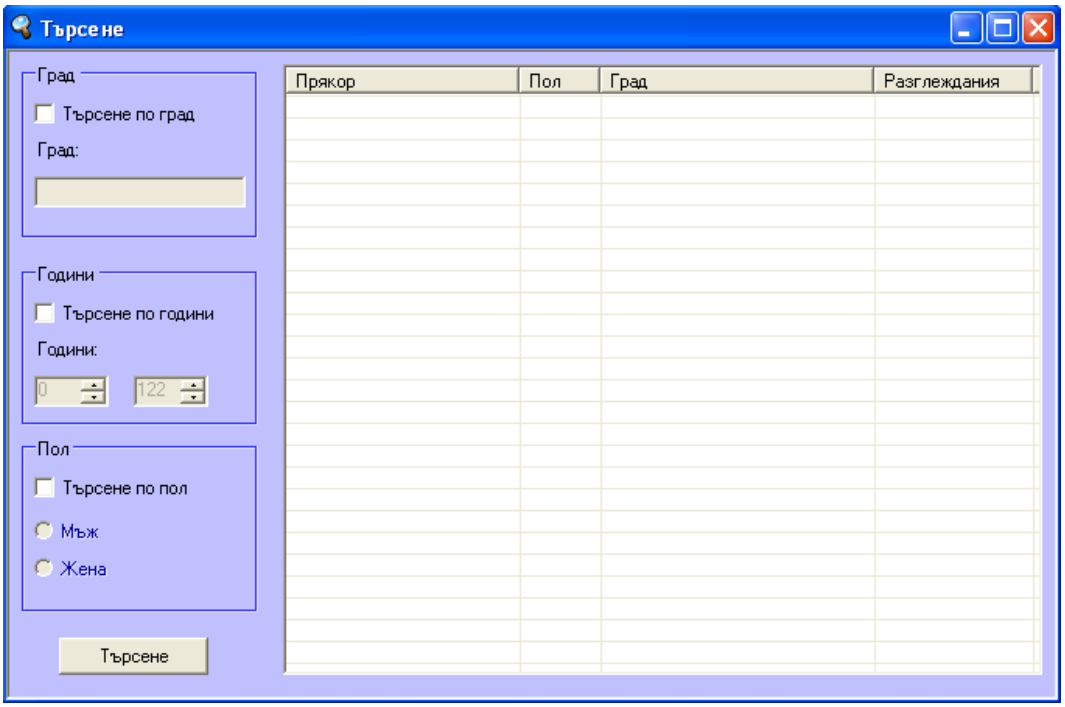
За визуализиране и въвеждане на съобщения ще използваме **`TextBox`** контрола. В случая се нуждаем **`TextBox`** контролата да е на много редове. За това ще променим свойството **`Multiline`** на **`true`**.

При воденето на разгорещен разговор между двама потребители би било удобно да има клавишна комбинация за изпращане на въведено съобщение – нека съобщението да се изпраща при натискане на клавиша **[Enter]**. За целта е необходимо да се "абонираме" за събитието **`KeyPressed`** на контролата **`TextBoxSend`**:

```
private void TextBoxSend_KeyPress(object sender,
     System.Windows.Forms.KeyPressEventArgs e)
{
   if( e.KeyChar == (char) Keys.Return)
   {
      SendMessage();
   }
}
```

Форма за търсене

При изграждане на формата за търсене трябва да обърнем внимание на няколко основни момента. Трябва да предоставим интерфейс, показващ недвусмислено дали търсенето включва даден критерий за търсене или не. За целта ще използваме **`CheckBox`** бутони.



Приели сме най-голямата възраст за търсене да е 122 години (според рекордите на Гинес, не е имало по-възрастен човек).

Нуждаем се да имплементираме функционалност за добавяне на потребител в "списък с приятели". Ще използваме същия подход както при преместването на приятели от една категория в друга. Ще създадем едно динамично меню.

Клиентски слой – ASP.NET уеб приложението

Целта на уеб приложението е да предостави лесно достъпен от Интернет потребителски интерфейс за работа с основната функционалност на системата, както е описана в нейната спецификация. Ще изградим този уеб базиран потребителски интерфейс със стандартните средства на ASP.NET и чрез използване на бизнес слоя от уеб услуги, който вече дискутирахме.

Дизайн съображения, свързани с уеб приложението

При реализацията на целите на приложението се използват основно идеите на шаблона за дизайн *Model–View-Controller (MVC)*, пречупен през характеристиките на едно уеб приложение.

Накратко казано, шаблоните за дизайн описват същината на решението на често срещани проблеми, възникнали при реализирането на едно приложение. По този начин не ни се налага всеки път, като се появи подобен проблем да преоткриваме решението му, а е нужно просто да изберем и приложим подходящ шаблон в контекста на конкретната задача.

Сега ще разгледаме шаблона за дизайн *MVC*, който споменахме по-горе. Той е изключително често използван при реализацията на потребителския интерфейс на едно приложение. Поради това, неговите концепции ще ни помогнат да моделираме функционалността на нашето уеб приложение, като същевременно с това повишим гъвкавостта и преизползваемостта на получената реализация.

Шаблон за дизайн Model-View-Controller (MVC)

MVC се състои от три основни компонента:

1. *Model* - отговаря за абстракцията на данните в приложението;
2. *View* - отговаря за визуалната презентация на приложението;
3. *Controller* - отговаря за реакцията на потребителския интерфейс към потребителските команди.

Разделението на логиките ни дава възможност да променяме всяка една от тях, независимо от останалите, което ни осигурява по-голяма гъвкавост. Например, ако сменим конкретната визуална презентация на данните, ще се наложи да променим само *View* компонента, а останалите части ще останат същите.

Поради това, че *Model* и *View* компонентите нямат директна връзка, имаме възможност по едно и също време да поддържаме различни визии на един и същи модел, т.е. едни и същи данни можем да ги показваме по различни начини на потребителя.

Друго предимство на този шаблон е това, че трите компонента могат до известна степен да съществуват независимо един от друг, което повишава тяхната преизползваемост.

MVC е фундаментален шаблон за дизайн, поради това той е в основата на други шаблони и има множество различни варианти и модификации.

Изграждане на страниците

Шаблонът за дизайн, който най-добре съчетава *MVC* подхода и същевременно с това е лесно приложим в изграждането на уеб приложения е *Page Controller*. Неговата основна идея е да получи HTTP заявката, да извлече необходимата му информация от нея, да извика дадени методи от бизнес логиката, и в зависимост от резултатите да определи вида на изходната страница. Именно този подход се използва при изграждането на страниците на приложението, като реализацията му чрез ASP.NET е изключително проста поради наличието на абстракцията **`System.Web.UI.Page`**. Тя ни предоставя изцяло базиран на събития начин за осъщест-

вяване на споменатите идеи абстрахирайки се от конкретното им предаване в HTTP средата.

Други основни правила, спазвани при изграждането на страниците, са:

1. Пълно разделение на логиката, отговаряща за управление на страницата от тази, отговаряща за нейната визия чрез използването на code behind.
2. Самостоятелната функционалност и визия, която може да бъде преизползвана в рамките на различни страници, се енкапсулира в потребителски контроли.

Решение за достъпа до уеб услугата

Генерирането на прокси класа за достъп до уеб услугата и прокси класовете на **`DTO`** и **`Status Enum`**, използвани от нейните методи, се осъществява автоматично от средата за разработка при добавяне на услугата.

Достъпът до услугата ще реализираме през клас, който обгръща нейното прокси, предоставяйки същия интерфейс (подобно на *Decorator* шаблона, който реализира добавяне на допълнителна функционалност към клас чрез обгръщане на конкретен негов екземпляр, а не наследяване). Целта му е да получи резултата от изпълнението на даден метод, да обработи статуса и ако той абстрахира изключителна ситуация, възникнала в уеб услугата, да предизвика съответното изключение, което да се разпространява в презентационния слой. По този начин Page Controller класовете се абстрахират от конкретната реализация на предаване на изключителните ситуации през уеб услугата и получават по-удобен механизъм за обработването им.

Автентикация на потребителите

За автентикация на потребителите ще използваме вградената в ASP.NET технология "Forms Authentication".

За да повишим сигурността, всички пароли ще предаваме в хеширан вид (по алгоритъм MD5), като хешрането ще извършваме още при клиента.

За реализиране на различията между групите потребители в системата ще използваме Role Based Security. Чрез него на всеки автентикиран посетител ще съпоставяме роля на администратор или потребител.

Установяването на идентичността на посетителя и неговата роля ще осъществяваме в уеб услугата. Тази информация се съхранява в cookie при клиента, което е стандартен подход при използване на такъв вид автентикация.

Управление на сесиите

При условие, че използваме технологията "Forms Authentication", която е базирана на cookie, осигуряващо автентичността на посетителите, то можем спокойно да съхраняваме и сесийния идентификатор в in-memory

cookie при клиента. Също така приложението използва стандартния механизъм за управление на **`Session`** обекта - **`InProc`**.

Както вече бе споменато, уеб услугата реализира собствен механизъм за управление на сесиите, базиран на таен низ, който тя предоставя при автентикация, а ние подаваме този низ на всеки неин метод, който го изисква. Съхранението на този низ е в сесийния обект от съображения за сигурност.

Поддръжката на собствена сесия от услугата налага разработването на начин за синхронизация на двете сесии. Този процес трябва да се извършва при всяка автентикирана заявка, за която и да е страница.

Управление на изключителните ситуации и грешките

Всички грешки, възникнали в приложението ще абстрахираме чрез изключения. Когато възникне изключителна ситуация, ако може тя се обработва от метода. Необработените от метода изключения ще се оставят да се разпространят по стека без да се обгръщат. Причината за това е в малката по обем и разслоеност презентационна логика, което намалява ползата от обгръщането на изключенията в такива, които да са по-смислени за извикващите методи.

Всички необработени изключения ще се обработват чрез механизма за глобална обработка на изключенията.

Ще реализираме записването на всички изключителни ситуации в exception log файл, който се намира в **`temp`** директорията на потребителя (user account), с който работи ASP.NET приложението. Причината за това решение е, че **`temp`** директорията заедно с директорията **`temporary files`** са единствените места по файловата система, за които стандартният потребител, с който работи ASP.NET, има права да пише. Другото удобно място за записване на изключения е Event Log, за който стандартният потребител също няма права за писане.

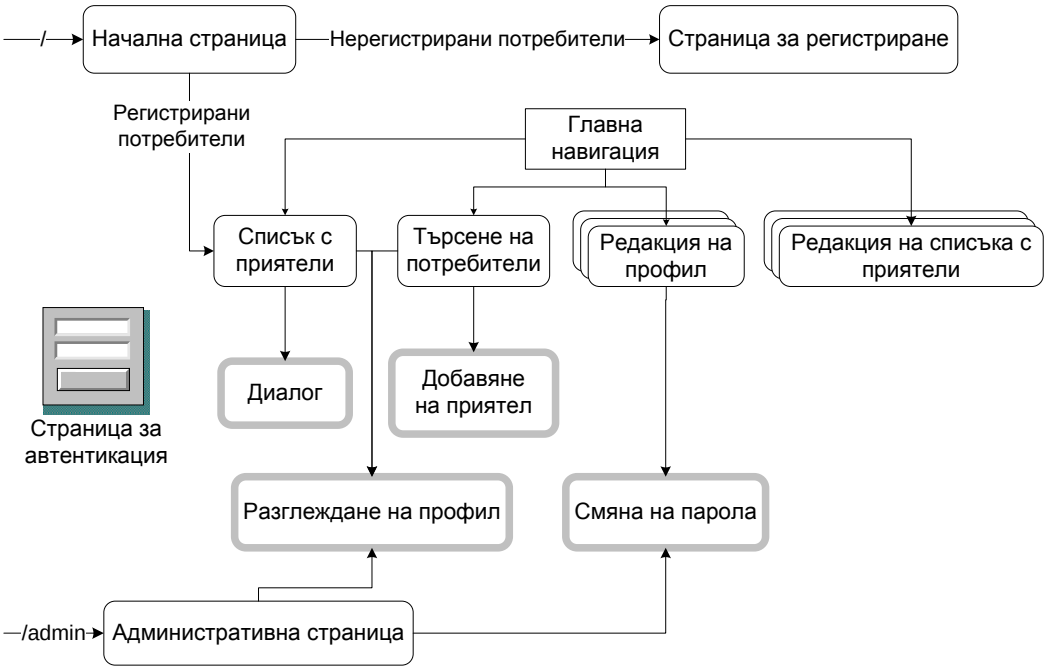
Поради трислойната архитектура на системата достъпът до базата данни може да мине само през уеб услугата, което не е добра практика за записване на изключения. Ако искаме да избегнем ограниченията на стандартния потребител, може да се имперсонираме временно (в рамките на метода, в който извършваме процедурата по записване на изключенията) с потребител, който има необходимите права, но за изискванията на това приложение подобно решение не е нужно.

Имплементация на ASP.NET уеб приложението

Нека сега разгледаме основните моменти от реализацията на ASP.NET уеб приложението.

Диаграма на уеб приложението

Ето как изглежда диаграмата на страниците в уеб приложението и преходите между тях (screen flow diagram):



Уеб приложението обслужва три групи посетители: нерегистрирани потребители, регистрирани потребители и администратор. Поради тази причина в сайта се образуват четири основни вида страници в зависимост от групите, които имат достъп до тях:

1. Страници, достъпни само за регистрираните потребители.
2. Страници, достъпни само за администратора на системата.
3. Страници, до които има достъп както администратора, така и потребителите.
4. Страници, достъпни за всички посетители на сайта.

Изграждане на страниците

Сега ще се спрем на основните концепции, свързани с реализирането на всяка една страница от приложението.

Всички контролери на страниците ще наследяват класа **`BaseDatingSitePage`**, който ще има за цел да дефинира базовия контролер за текущото приложение. Неговата реализация ще е доста абстрактна, като основното нещо, което ще дефинира е един *Template Method* – **`InitializeComponent`**, който ще се предефинира от конкретния контролер и в него ще бъде реализирана логиката за инициализация на компонентите (най-често абониране за събития).

```
protected override void OnInit(EventArgs e)
{
   InitializeComponent();
```

```
    base.OnInit(e);
}

protected virtual void InitializeComponent()
{
}
```

По отношение на визията на страниците във всички тях ще се включва една потребителска контрола, която ще дефинира съдържанието на **<head>** секцията на **html**, а именно мета информацията, **css** файла със стилове, **popup.js** файла и ще се параметризира заглавието и. Освен това и в страниците с главна навигация ще се включва потребителска контрола, която реализира нейната визия.

За конкретен пример ще проследим изграждането на страницата **edit_friends_list.aspx**.

Нейната цел е да предостави на потребителя възможност да редактира своя списък с приятели. За това първо трябва да намерим начин за визуализация на списъка с приятели. При търсене на правилния подход е хубаво да имаме предвид, че идеите му могат да бъдат преизползвани и в страницата за преглед на списъка.

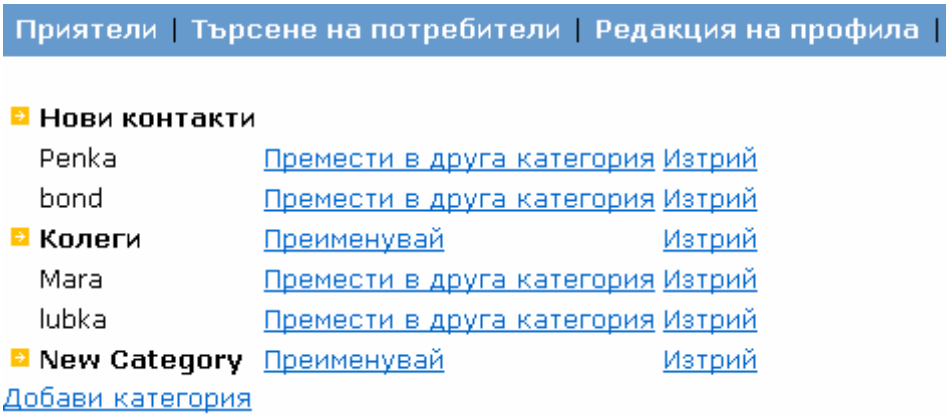
Изграждане на списъка с приятели

Списъкът с приятели представлява йерархична структура с едно ниво на вложеност. Това ни навежда на мисълта, че е удобно използването на вложени контроли за реализацията му. Поради своята лекота и възможност за по-пълен контрол върху **html**, който се продуцира, **Repeater** контролата е подходящ кандидат за тази цел. Сега остава да реализираме попълването и с данни. Това се случва в метода **Page_Load**, като ако страницата не е в състояние на **PostBack** се извиква метода **PopulateFriendsList()**, който попълва списъка. Това става, като на външния **Repeater** се подаде за **DataSource** масив с обекти от тип **CategorizedFriendsDTO**. След това в метода "абониран" за неговото събитие **ItemDataBound**, на съответния вътрешен **Repeater** подаваме за **DataSource** масива с приятелите в текущата категория.

```
private void PopulateFriendsList()
{
   GetCategorizedFriendsResult result =
     DatingSiteServiceProvider.GetCategorizedFriendsList(UserID,
        WebServiceSessionID);
   RepeaterCategories.DataSource = result.CategorizedFriends;
   RepeaterCategories.DataBind();
}

private void RepeaterCategories_ItemDataBound(object sender,
    RepeaterItemEventArgs e)
```

```
{
   CategorizedFriendsDTO categorizedFriends =
      (CategorizedFriendsDTO) e.Item.DataItem;

         ...

   Repeater RepeaterFriends =
      e.Item.FindControl("RepeaterFriends") as Repeater;
   if (RepeaterFriends != null)
   {
      RepeaterFriends.DataSource = categorizedFriends.Friends;
      RepeaterFriends.DataBind();
   }
}
```

Ето как изглежда страницата, която показва списъка с приятели:



Сега, нека разгледаме реализирането на операциите, свързани с редакцията на списъка с потребителите.

Операции за редакция на списъка

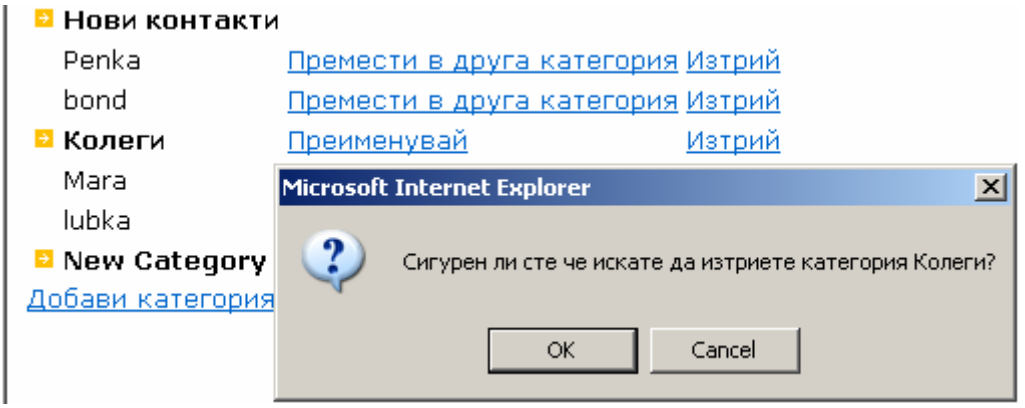
Това са действията за добавяне, преименуване, изтриване на категория, преместване и изтриване на приятел от дадена категория. В зависимост от нуждите на потребителя можем да разделим действията на два типа:

- действия, изискващи само потвърждение (изтриване);
- действия, изискващи допълнителна информация свързана с осъществяването си (добавяне, преименуване и преместване).

Поради това разделение ще използваме различни подходи за реализация на различните групи действия. Абстрактният критерий, използван за отделяне ни кара и в двата случая да търсим подход, удобен за преизползване на функционалността и в други страници.

Действия, изискващи само потвърждение

Ето как изглеждат действията, които потребителят трябва да потвърди преди изпълнението им:



Изключително удобен и лесен начин за реализация на този вид взаимодействия е чрез използването на JavaScript за извикване на диалога за потвърждение на съответния браузър при събитието **onclick** на **LinkButton**, отговарящ за изтриването. Поради съображения за преизползване ще реализираме контрола, която наследява **LinkButton** и осъществява съответната функционалност:

```
public class ConfirmationLinkButton : LinkButton
{
   public string ConfirmationMessage
   {
      set
      {
         this.Attributes["onclick"] = String.Format(
             "return confirm('{0}');", value);
      }
   }
}
```

По този начин само при положителен отговор на потребителя, ще се предизвика съответното **PostBack** събитие, отразяващо натискането на бутона, което ще се обработи от съответния абониран метод в контролера на страницата.

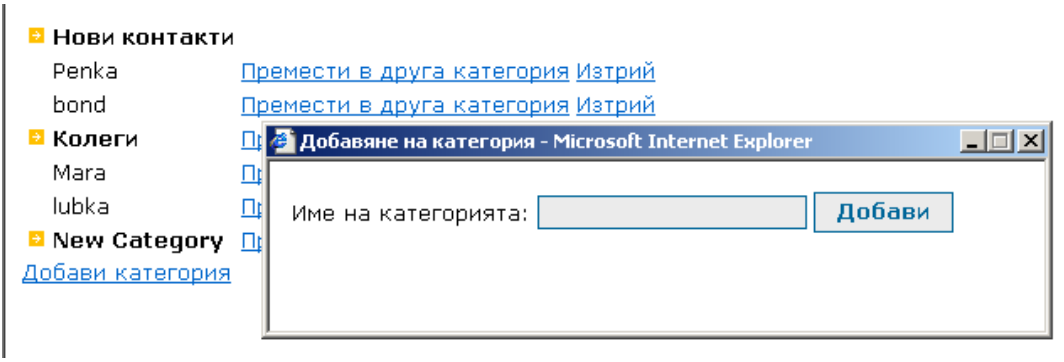
Конкретно в страницата за редакция това ще става чрез прихващане на събитието **ItemCommand** на съответния **Repeater**, където в зависимост от името на командата на съответния бутон се извиква методът за осъществяването й.

```
private void RepeaterCategories_ItemCommand(object source,
   RepeaterCommandEventArgs e)
{
```

```
    if (e.CommandName.Equals("DeleteCategory"))
    {
        int categoryID = Convert.ToInt32(e.CommandArgument);
        DeleteCategory(categoryID);
    }
}
```

Действия, изискващи допълнителна информация

Другата категория действия, които уеб приложението позволява на потребителите да извършват са действия, изискващи допълнителна информация за осъществяването си. Ето пример:



Тази група действия удобно се реализира чрез изнасяне на събирането на допълнителна информация извън основната страница чрез popup прозорец, който тя контролира.

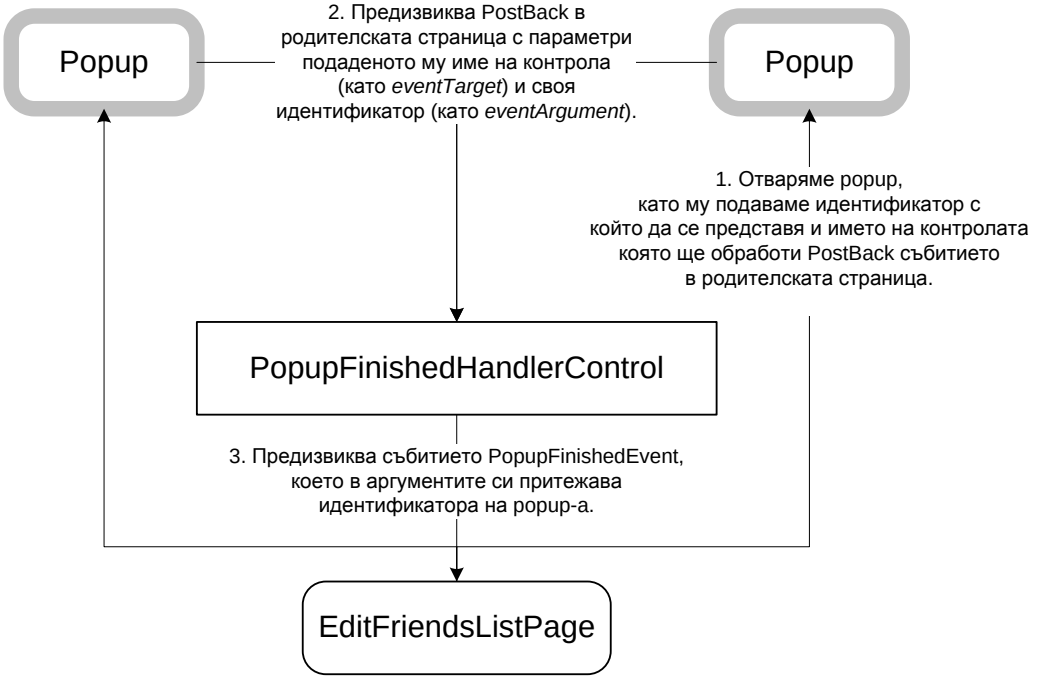
За работа с popup прозорци няма стандартна функционалност в ASP.NET и затова ще се наложи да измислим собствен механизъм за показване на popup диалози чрез JavaScript и за предаване на данните към основната страница, която да ги обработи.

Поради необходимостта основната страница да разбере кога даденото действие се е осъществило, трябва да реализираме възможност popup прозорецът да предизвиква **`PostBack`** събитие в отварящата го страница след като завърши работата си.

Както се вижда от схемата по-долу, за да осъществим тази комуникация, трябва да преминем през три етапа:

1. Отваряне на popup прозореца.
2. Предизвикване на **`PostBack`** събитие в отварящата страница.
3. Обработване на **`PostBack`** събитието.

Следващата схема илюстрира тези етапи:



Нека разгледаме описаните 3 етапа за работата с popup прозорци в по-големи детайли:

1. Отваряне на popup прозореца.

Когато отваряме popup прозореца чрез **QueryString** частта от неговия URL адрес му подаваме името на контролата (**handlerControl**), която ще обработи **PostBack** събитието и идентификатор (**popup_name**), с който той ще се идентифицира пред отварящата го страница.

```
var popup =
   window.open("../friendslist/rename_category.aspx?category_id="
   + categoryID + "&finished_handler=" + handlerControl +
   "&popup_name=" + popupName,"renameCategoryWindow",features);
```

2. Предизвикване на **PostBack** събитие в отварящата страница.

Отново с оглед на преизползване на тази функционалност, ще капсулираме това действие в отделна потребителска контрола **PopupFinishedDispatcherControl**. Тя ще взема съответната информация от **QueryString** и ще предизвиква **PostBack** в отварящата страница с параметри името на съответния **handler** и името на popup диалога.

```
window.opener.__doPostBack('<%= FinishedHandlerControl%>',
   '<%= PopupName%>');
```

3. Обработване на **PostBack** събитието.

Аналогично на предизвикването, обработката на **PostBack** събитието ще реализираме чрез потребителска контрола. Тя ще имплементира интерфейса **IPostBackEventHandler** и в дефинирания от него метод ще предизвиква нормалното събитие **PopupFinished** със съответните аргументи:

```
public class PopupFinishedHandlerControl : UserControl,
IPostBackEventHandler
{
   public event PopupFinishedHandler PopupFinished;

   protected PlaceHolder PlaceHolderPostbackEventReference;

   public void RaisePostBackEvent(string eventArgument)
   {
      if (PopupFinished != null)
      {
         PopupFinished(this,
            new PopupFinishedEventArgs(eventArgument));
      }
   }

   protected override void Render(HtmlTextWriter writer)
   {
      HtmlAnchor htmlAnchorPostBackReference =
        new HtmlAnchor();
      htmlAnchorPostBackReference.ID = this.UniqueID;
      htmlAnchorPostBackReference.HRef =
          String.Concat("javascript:",
             Page.GetPostBackEventReference(this));
      htmlAnchorPostBackReference.InnerHtml = this.UniqueID;
      PlaceHolderPostbackEventReference.Controls.Add(
          htmlAnchorPostBackReference);

      base.Render(writer);
   }
}

public class PopupFinishedEventArgs : EventArgs
{
   private string mPopupName;

   public string PopupName
   {
      get
      {
         return mPopupName;
      }
   }

   public PopupFinishedEventArgs(string aPopupName)
```

```
   {
      mPopupName = aPopupName;
   }
}

public delegate void PopupFinishedHandler(object sender,
   PopupFinishedEventArgs e);
```

Реализацията на този клас е стандартна, с изключение на метода **`Render()`**. Целта на предефинирането му е да задействаме включването на client-side елементите, реализиращи функционалността за **`PostBack`** в резултатния HTML на съдържащата контролата страница.

Сега остава само основната страница да консумира събитието, отразяващо успешното приключване на действието и да извърши необходимата в този случай обработка. Конкретно при страницата за редакция това е обновяването на списъка.

```
private void IncludePopupFinishedHandler_PopupFinished(
   object sender, PopupFinishedEventArgs e)
{
   PopulateFriendsList();
}
```

Визуализация на резултатите от търсенето

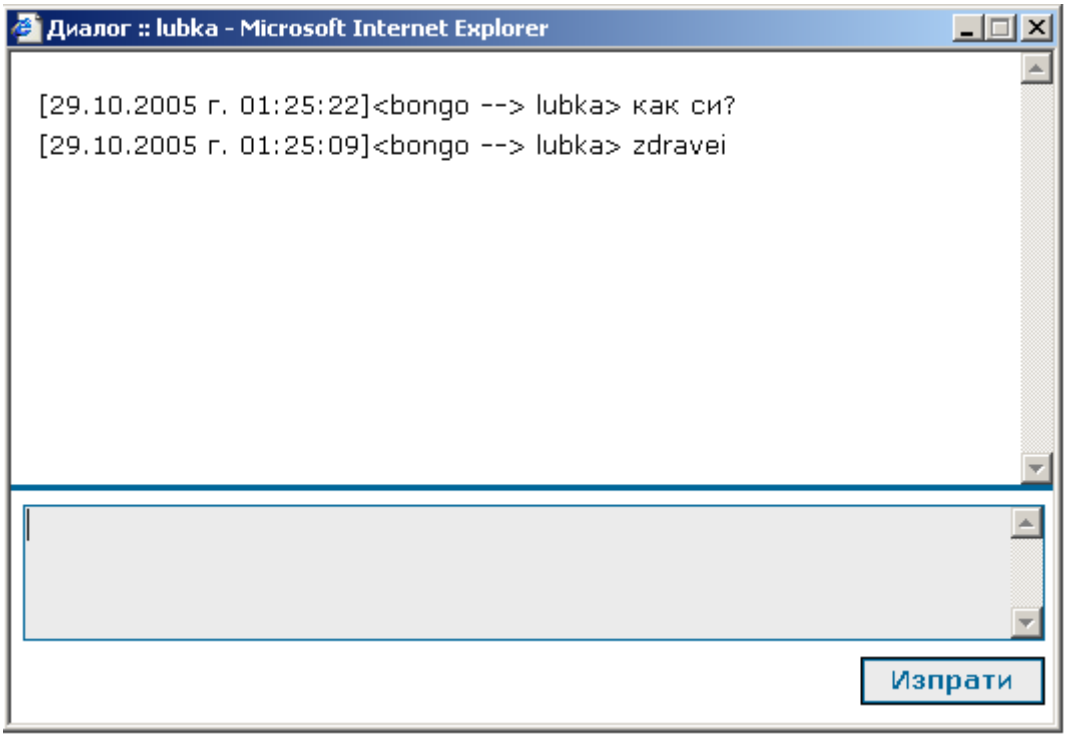
За визуализиране на намерените потребители от дадено търсене използваме **`DataGrid`** със страниране. Реализацията на страницирането извършваме като първо от услугата вземаме броя на намерените потребители, за да изчислим броя на страниците, а след това вземаме данните само на тези потребители, които са в рамките на текущо избраната страница.

Както бе споменато, за транспорт на данните между уеб услугата и ASP.NET приложението се използват Data Transport Objects. Прокси класовете на тези обекти, както и самите те, са класове с публични полета, което прави невъзможно използването на метода **`DataBinder.Eval()`** за извличане на стойностите на тези полета. **`BoundColumn`** използва именно този начин. Това заедно с липсата на **`HtmlEncoding`** на данните в нея ни кара да я заменим с **`TemplateColumn`**. Реализирането на това решение стана, като използваме **`ItemTemplate`** шаблона на колоната и в него просто включваме съответната информация в желания от нас вид:

```
<asp:templatecolumn headertext="Град">
   <itemtemplate>
   <%#
     Server.HtmlEncode(((UserProfileDTO)Container.DataItem).Town)
   %>
   </itemtemplate>
</asp:templatecolumn>
```

Реализация на диалога между потребители

Диалогът между потребители ще реализираме в popup страница, която изглежда по следния начин:



Поради функционалното изискване един потребител да има възможност да бъде в режим на диалог с няколко свои приятели едновременно, се налага при образуването на името на прозореца за диалог да участва и потребителският идентификатор на съответния приятел. Ето как реализираме това:

```
function openDialogPopup(userID, userName){
   ...
   var popup = window.open("../messages/dialog.aspx?user_id=" +
     userID + "&user_name=" + userName,
     "dialogWindow" + userID, features);
   if (window.focus) {popup.focus()}
}
```

Другото предимство на този метод е свойството, че когато потребителят се опита два пъти да влезе в режим на диалог с един и същ свой приятел съответния прозорец ще се преизползва т.е. няма да се отворят два отделни прозореца.

Самата страница за диалог ще реализираме като съвкупност от две страници, който са разположени в отделни рамки (frames). Едната страница **message_input.aspx** ще съдържа интерфейса за писане на съоб-

щение, а другата **messages_display.aspx** ще отговаря за визуализирането на последните 50 съобщения, разменени между двамата потребители. Този подход ни дава възможност да обновяваме двете страници по отделно. Така страницата **messages_display.aspx** се презарежда независимо на всеки 5 секунди, както и при успешно изпращане на съобщение от потребителя. За негово удобство се налага при всяко обновяване да се запазва текущата позиция на плъзгача на страницата. Това ще реализираме като при презареждането на страницата чрез JavaScript записваме текущите позиции и ги изпращаме към сървъра. След което при зареждане на обновената страница ги прилагаме отново върху нея.

```
function refreshPage(){
   persistScrollPositions();
   document.forms['FormMessages'].submit();
}

function applyScrollPosition(){
   window.scrollTo(<%= ScrollXPosition%>, <%= ScrollYPosition%>);
}

window.onload = applyScrollPosition;
window.setTimeout("refreshPage()", 5000);
```

В посочения код **ScrollXPosition** и **ScrollYPosition** са свойства на класа **MessagesDisplayPage**. Ето техните дефиниции:

```
protected string ScrollXPosition
{
   get
   {
      string result;
      if (!HtmlInputHiddenScrollXPosition.Value.Equals(
           String.Empty))
      {
         result = HtmlInputHiddenScrollXPosition.Value;
      }
      else
      {
         result = "0";
      }
      return result;
   }
}

protected string ScrollYPosition
{
   get
   {
      string result;
```

```
    if (!HtmlInputHiddenScrollYPosition.Value.Equals
          (String.Empty))
    {
      result = HtmlInputHiddenScrollYPosition.Value;
    }
    else
    {
      result = "0";
    }
    return result;
  }
}
```

JavaScript методът **`persistScrollPositions()`** взема текущите позиции на плъзгача и ги записва в скрити полетата на формата **`FormMessages`**.

Визуализирано на съобщенията реализираме чрез **`Repeater`** контрола.

Реализация на достъпа до уеб услугата

Както споменахме в предната част, достъпът до уеб услугата ще осъществим чрез клас **`DatingSiteServiceProvider`**, който я обгръща и прави проверка за вида на статуса. Поради това, че услугата не се променя в рамките на едно стартиране на приложението, е хубаво да спестим безсмисленото създаване на нов екземпляр от нея при всяко викане на неин метод. Това ще постигнем като направим всички методи на класа статични и използваме статична член-променлива, която сочи към екземпляр на услугата, създаден в статичния конструктор.

```
public class DatingSiteServiceProvider
{
  private static DatingSiteWebService mService =
      new DatingSiteWebService();
  ...
```

Методът, който ще извършва съответната проверка, е **`ThrowIfException()`**. Той ще използва факта, че всички статуси, които абстрахират изключения в контекста на повечето методи са описани в **`MethodStatus`**.

```
private static void ThrowIfException(String aStatus)
{
  Exception exception = null;
  if (aStatus.Equals(MethodStatus.InvalidSession.ToString()))
  {
    exception = new InvalidSessionException();
  }
  else
  if (aStatus.Equals(MethodStatus.InvalidParameters.ToString()))
```

```
    {
      exception = new InvalidParametersException();
    }
    else
    if (aStatus.Equals
          (MethodStatus.InternalServerError.ToString()))
    {
        exception = new InternalServerErrorException();
    }
    else
    if (aStatus.Equals
          (MethodStatus.OperationNotPerformed.ToString()))
    {
      exception = new OperationNotPerformedException();
    }
    if (exception != null)
    {
      throw exception;
    }
}
```

Изключение от това правило правят само методите **`UpdateLastActivity()`** и **`RefreshAdminSession()`**, в чиито контекст статусът **`InvalidSession`** не е изключение, а статус на резултат. Поради това обработката на техния статус ще изнесем в друг метод.

Реализация на класа SiteUtility

При реализацията на дизайн съображенията, свързани с автентикацията на потребителите и управлението на сесиите, се налага в приложението да реализираме логика, която не е свързана директно с конкретно изпълнение на функционалните изисквания на приложението. Тази логика е добре да бъде енкапсулирана в един клас с цел да абстрахираме контролерите на страниците и контролера на приложението от нейната конкретна реализация. За тази цел ще създадем класа **`SiteUtility`**.

За да осъществим логиката, свързана с управлението на идентичността на автентикирания потребител, неговата роля и низа му за автентикация пред услугата ще реализираме три метода:

1. Методът **`LogIn()`**

Той ще се извиква при автентикация на потребителя от формата за вход в системата и ще отговаря за съхранението на идентичността на автентикирания потребител, неговата роля и низ между отделните HTTP заявки. Тази функционалност ще реализираме чрез записване на съответната информация в cookie, което после от съображения за сигурност ще криптираме.

```
public static void LogIn(int aUserID, string aRole,
```

```
   string aWebServiceSessionID)
{
   DatingSiteSessionUtility.WebServiceSessionID =
      aWebServiceSessionID;

   FormsAuthentication.Initialize();

   FormsAuthenticationTicket ticket =
      new  FormsAuthenticationTicket(1,
         Convert.ToString(aUserID),
         DateTime.Now,
         DateTime.Now.AddMinutes(20),
         false,
         aRole,
         FormsAuthentication.FormsCookiePath);

   string encryptedTicket = FormsAuthentication.Encrypt(ticket);

   HttpCookie cookie = new
      HttpCookie(FormsAuthentication.FormsCookieName,
      encryptedTicket);
   cookie.Expires = ticket.Expiration;
   HttpContext.Current.Response.Cookies.Add(cookie);
}
```

2. Методът **`CreatePrinciple()`**

Поради природата на HTTP протокола се налага след всяко успешно автентикиране на заявка да извличаме ролите от бисквитката и да създаваме наново **`Principle`** обекта на текущия контекст. Именно тази функционалност ще реализираме чрез метода **`CreatePrincipal()`**:

```
public static IPrincipal CreatePrincipal()
{
   FormsIdentity identity =
      (FormsIdentity) HttpContext.Current.User.Identity;
   FormsAuthenticationTicket ticket = identity.Ticket;
   string role = ticket.UserData;
   GenericPrincipal result =
      new GenericPrincipal(identity, new string[] {role});
   return result;
}
```

Този метод ще се извиква в **`Global.asax`** файла от метода **`Application_AuthenticateRequest()`**.

```
protected void Application_AuthenticateRequest(Object sender,
     EventArgs e)
{
   if (HttpContext.Current.User != null &&
```

```
      HttpContext.Current.User.Identity.IsAuthenticated)
   {
      HttpContext.Current.User = SiteUtility.CreatePrincipal();
   }
}
```

3. Методът **`LogOut()`**

Целта му е да реализира излизането от системата (logout). Ето как е реализирана тази функционалност:

```
public static void LogOut()
{
   if (DatingSiteSessionUtility.WebServiceSessionID != null)
   {
      DatingSiteServiceProvider.LogOut(
          DatingSiteSessionUtility.WebServiceSessionID);
   }
   DatingSiteSessionUtility.ClearWebServiceSessionID();
   FormsAuthentication.SignOut();
}
```

Съответно логиката, свързана със синхронизацията на сесиите ще реализираме в метода **`SynchronizeSessions()`**. Той също ще се изпълнява при всяка автентикирана заявка и ще се извиква в **`Global.asax`** файла от метода **`Application_AcquireRequestState()`**. Причината да се случва тогава е в това, че низът за автентикация пред услугата се съхранява в **`Session`** обекта, а това е първият метод, при който той е достъпен. Ето как изглежда реализацията му:

```
public static void SynchronizeSessions()
{
   MethodStatus result;
   if (DatingSiteSessionUtility.WebServiceSessionID != null)
   {
      if (HttpContext.Current.User.IsInRole(
        DatingSiteRoles.ADMIN))
      {
      result = DatingSiteServiceProvider.RefreshAdminSession(
        Convert.ToInt32(HttpContext.Current.User.Identity.Name),
        DatingSiteSessionUtility.WebServiceSessionID);
      }
      else
      if (HttpContext.Current.User.IsInRole(
        DatingSiteRoles.USER))
      {
      result = DatingSiteServiceProvider.UpdateLastActivity(
        Convert.ToInt32(HttpContext.Current.User.Identity.Name),
        DatingSiteSessionUtility.WebServiceSessionID);
```

```
      }
      else
      {
         result = MethodStatus.InvalidSession;
      }
   }
   else
   {
      result = MethodStatus.InvalidSession;
   }

   if (result == MethodStatus.InvalidSession)
   {
      SiteUtility.RedirectToLoginPage();
   }
}
```

Тази синхронизация ще се осъществява чрез извикването на два метода от услугата в зависимост от ролята на автентикирания посетител. Ако е администратор, ще се извиква методът **`RefreshAdminSession()`**, а ако е потребител – методът **`UpdateLastActivity()`**, който освен да обновява сесията има за цел и да записва датата и часа на текущата активност с цел реализиране на функционалността за следене на активност. Във всички останали случаи се приема, че сесията е невалидна и съответно заявката се прехвърля към страницата за автентикация.

Тази реализация на управлението на сесиите има и друго важно предимство. Тя се явява като филтър на всяка автентикирана заявка като гарантира, че автентикирания потребител на текущия контекст притежава валидна сесия. Благодарение на това, че съответния идентификатор на сесията се съхранява на сървъра в **`Session`** обекта (който от своя страна се намира само в паметта на приложението), се осигурява и защита от евентуална подмяна на информацията в cookie обекта, свързан с идентичността на потребителя. Причината е в това, че тази информация ще се верифицира при всяка автентикирана заявка от услугата, в зависимост от идентификатора на сесията.

Реализация на извличането на снимка на потребител

Тази функционалност ще осъществим чрез обработване на заявката от **`HttpHandler`**. Причината за това е, че резултатът от тази заявка не е HTML страница, а ресурс от тип **`image`**, и използването на Page Controller в този случай е неудачно. По същия начин ще постъпим и с останалите подобни страници: страницата за извличане на XML отчет и страницата за извличане на защитната картинка.

Реализирането на handler-а става чрез клас, имплементиращ интерфейса **`IHttpHandler`**, който дефинира метод **`ProcessRequest`** с параметър текущия контекст. Поради необходимостта за достъп до **`Session`** обекта от

текущата заявка, трябва да реализираме и маркиращия интерфейс **IRequiresSessionState**.

```
public class GetPictureHttpHandler : IHttpHandler,
     IRequiresSessionState
   {
     public const int BUFFER_SIZE = 10240;

     public void ProcessRequest(HttpContext context)
     {
       Stream data = null;
       try
       {
         HttpRequest request = context.Request;
         HttpResponse response = context.Response;

         response.BufferOutput = false;

         ...

         WebClient client = new WebClient();
         data = client.OpenRead(WebServiceUserPictureUrl);

         response.Clear();
         if (client.ResponseHeaders["Content-Type"] == null ||
           client.ResponseHeaders["Content-Type"].Equals(
           String.Empty))
         {
           response.ContentType = "image/jpeg";
         }
         else
         {
           response.ContentType =
             client.ResponseHeaders["Content-Type"];
         }

         byte[] buffer = new byte[BUFFER_SIZE];
         int bytesRecieved = 0;
         do
         {
           bytesRecieved = data.Read(buffer, 0, BUFFER_SIZE);
           response.OutputStream.Write(buffer, 0,
             bytesRecieved);
         } while (bytesRecieved > 0);
       }
       catch (Exception ex)
       {
         LoggingExceptionHandler.HandleException(ex);
       }
       finally
       {
```

```
            if (data != null)
            {
               data.Close();
            }
         }
      }
   }
```

Както бе обяснено в частта за уеб услугата, извличането на снимката на даден потребител трябва да става без тя да се буферира в паметта. За това указваме отговорът на заявката да не се буферира и чрез класа **WebClient** отваряме поток към **aspx** страницата от уеб услугата, отговаряща за предоставянето на снимката и на малки порции от 1024 байта предаваме информацията от този поток в потока на отговора.

Поради специфичния тип на съдържанието, което се връща при заявка към тази страница, обработката на изключителните ситуации не може да се извърши по същия начин, както при останалите страници. По конкретно, при очакван тип image на отговора е неприемливо при възникване на изключителна ситуация да върнем html страница за грешка. За тази цел ще прихващаме базовия клас изключение **Exception**, което, въпреки че не е добра практика, в тази ситуация е приемливо. След това прихванатото изключение само ще записваме в лога и няма да го прехвърляме нагоре по стека, защото в противен случай ще попадне в глобалния механизъм за обработка на изключения, а както казахме, в този случай това е неприемливо.

Пренасочване на отговора към друга страница

Тук ще се спрем върху едно по-различно решение свързано със специфичното поведение на методите **Response.Redirect()** и **Server.Transfer()**. Както е описано в http://support.microsoft.com/kb/312629/EN-US, тези методи вътрешно в себе си извикват метода **Response.End()**, който предизвиква **ThreadAbortException**. Често решение на този проблем е прихващането на изключението в празен **catch** блок, но ние ще използваме по-различно решение, препоръчано от Microsoft, което ще ни върши същата работа. Идеята е навсякъде да използваме **Response.Redirect(string, bool)**, като вторият параметър указва дали да бъде "сложен край на отговора". Вместо **Server.Transfer()** ще използваме **Server.Execute()**.

Инсталиране и внедряване на системата

След като разгледахме по-важните моменти от създаването на системата за запознанства в Интернет, ще дадем кратко описание на процеса на инсталацията ѝ. Препоръчваме на читателя да опита да инсталира системата на собствения си компютър, да разгледа нейния сорс код и да се опита да разбере как работи той. Системата представлява сериозен про-

ект, от който може да заимствате идеи и решения на технически проблеми при изграждането на подобни системи. Разбира се, не трябва всичко да копирате, а само това, което ви върши работа в конкретния случай.

Системни изисквания

За да инсталирате успешно системата, е необходимо да имате компютър с Windows 2000/XP/2003, IIS 5.0/6.0, SQL Server 2000, .NET Framework 1.1 и VS.NET 2003. Възможно е системата да работи и с други версии на посочения софтуер, но няма гаранция за това.

За да инсталирате системата са ви необходими администраторски права върху машината, която ще използвате.

От къде да изтеглим системата и сорс кода й?

Готовото приложение заедно с пълния изходен код можете да инсталирате чрез инсталационния пакет, достъпен от уеб сайта на настоящата книга: http://www.devbg.org/dotnetbook/.

Като алтернатива можете да инсталирате всеки компонент отделно, изтегляйки от същия сайт единен архивен файл, съдържащ 4 архива на различните компоненти:

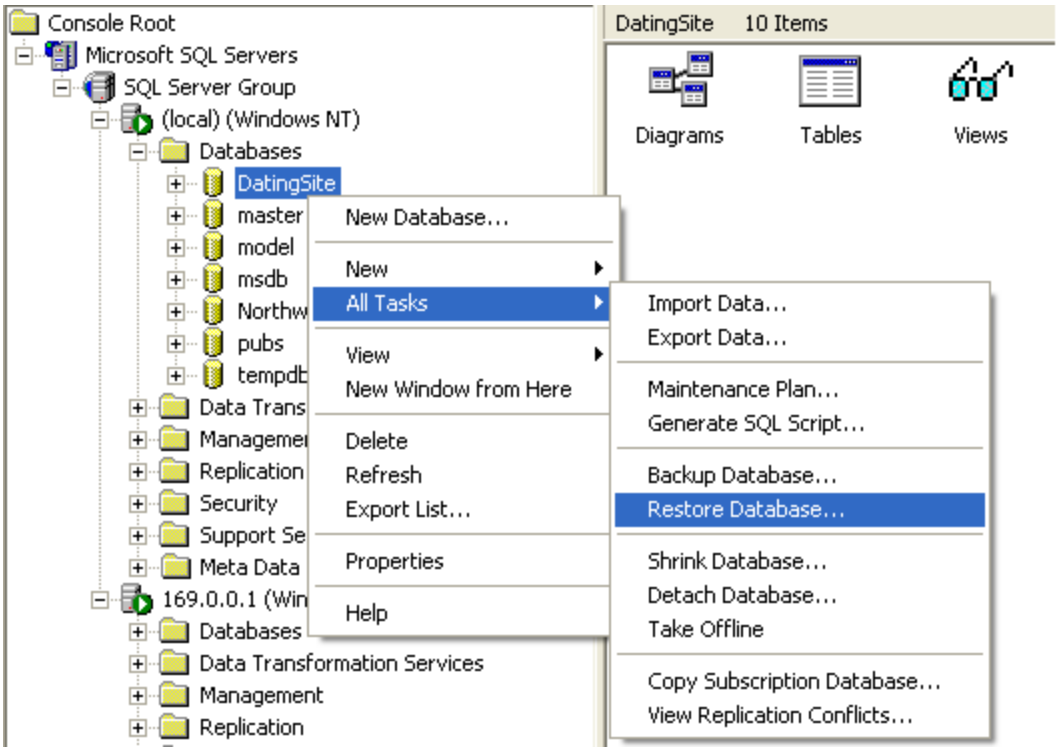
- копие на SQL Server базата данни (backup);
- проект с ASP.NET уеб услугата;
- проект с Windows Forms приложението;
- проект с ASP.NET уеб приложението.

Да разгледаме как се инсталира и конфигурира всеки един от тези компоненти.

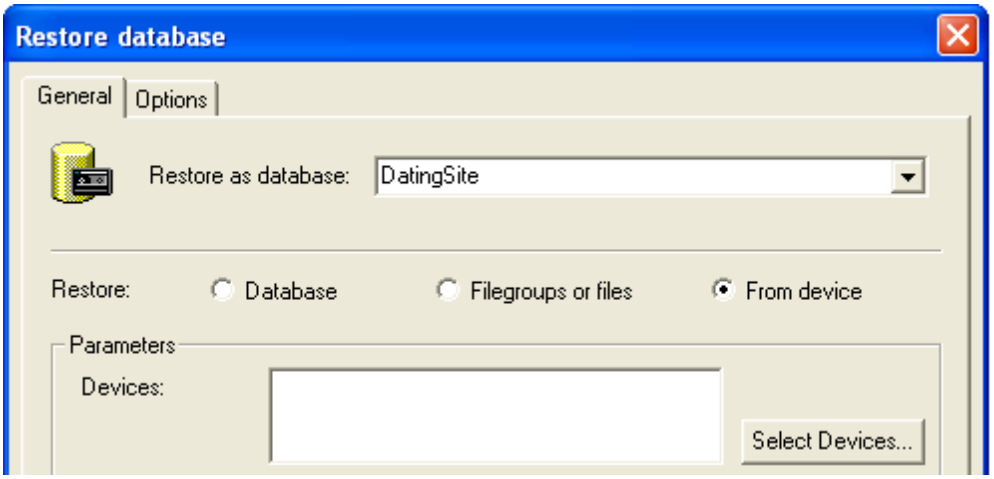
Възстановяване на базата данни в SQL Server

За инсталиране на базата данни във вашия SQL Server ви е необходим файла **`DatingSiteDB_backup`**, който представлява неин архив (backup). Ето и как се извършва процесът на възстановяване (restore) на базата данни:

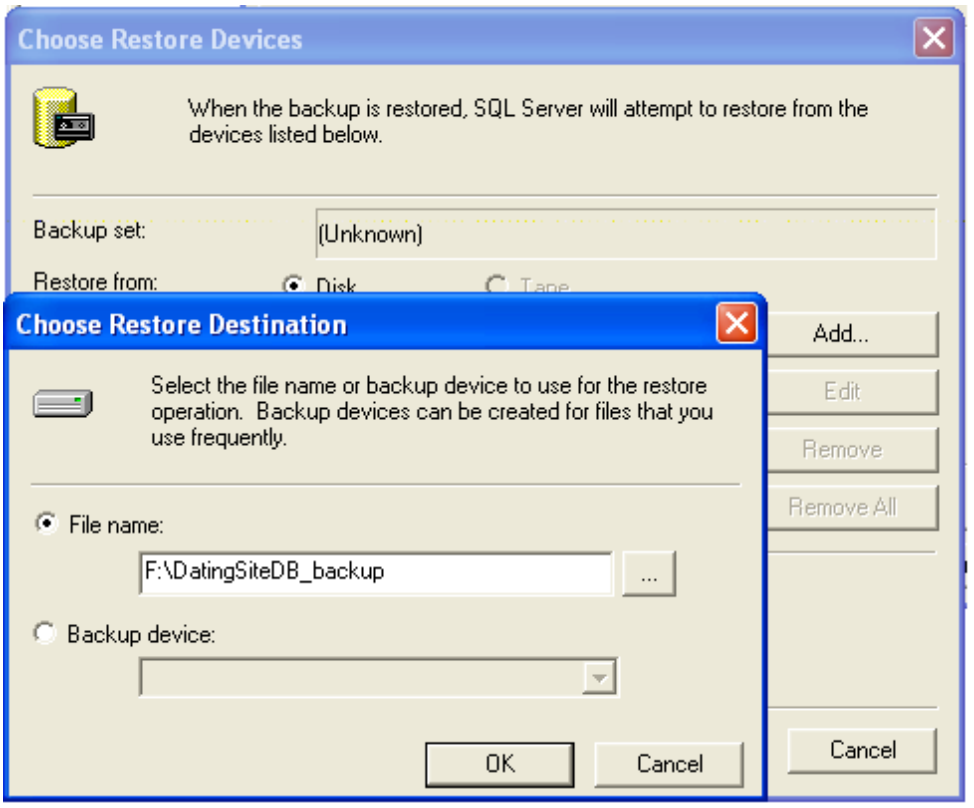
1. Отворете Enterprise Manager и създайте база от данни с име **`DatingSite`**.
2. Натиснете с десния клавиш на мишката върху нея и от popup менюто изберете **All Tasks -> Restore Database...**



3. От диалоговия прозорец изберете **[Restore: From Device]** и натиснете бутона **[Select Devices...]**:



4. Появява се нов диалогов прозорец, в който трябва да натиснете бутона **[Add]**. В новия прозорец трябва да изберете пътя до файла, съдържащ backup на базата данни. В нашия случай това е файлът **`DatingSiteDB_backup`**:



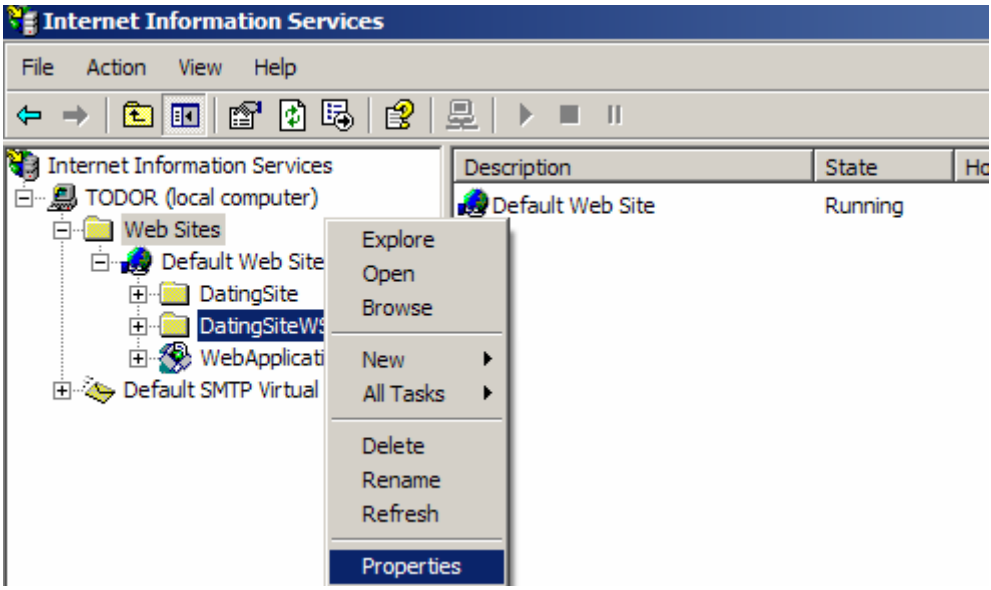
5. При натискане на бутона **[OK]** започва процесът на възстановяване на базата данни и след като той завърши успешно, внедряването на базата данни приключва.

Инсталиране и внедряване на ASP.NET уеб услугата

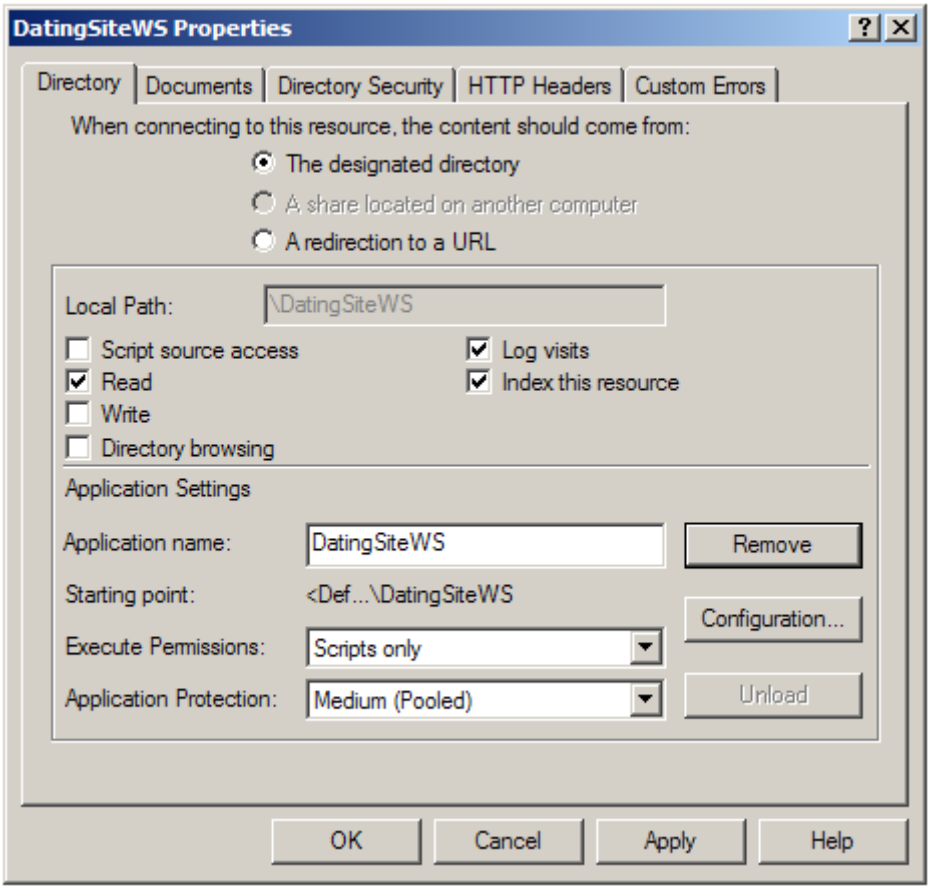
Уеб услугата може да бъде инсталирана върху всички платформи, подържащи ASP.NET. В настоящите инструкции ще покажем единствено внедряването ѝ върху Microsoft Internet Information Server 5.0.

Инсталиране на уеб услугата в IIS

1. Разархивирайте съдържанието на уеб услугата в директорията **`wwwroot`** на IIS, която по подразбиране се намира на **`C:\Inetpub\wwwroot`**.
2. Стартирайте конзолата за управление на IIS и натиснете дясно копче върху появилата се папка **`DatingSiteWS`**.
3. Изберете **`Properties`**.



4. Натиснете бутона **[Create]** в зоната **Application Settings** на появилия се прозорец. Накрая той трябва да изглежда така:



5. Натиснете **[OK]**.

Конфигуриране на уеб услугата

1. Отворете конфигурационния файл **`Web.config`**.
2. Променете стойността на полето **`siteDatabaseConnectionString`**, така че да се оказват валидни стойности за връзка към базата данни, която преди малко инсталирахте.
3. Променете стойността на полето **`siteSMTPServer`**, така че да оказва валиден SMTP сървър. Бихте могли да използвате SMTP сървъра на вашия Интернет доставчик. Ако SMTP сървъра използва потребителско име и парола за автентикиране на потребители бихте могли да окажете такива чрез полетата **`siteSMTPServerUserName`** и **`siteSMTPServerPassword`**.

Инсталиране на Windows Forms клиента

Стъпките за инсталиране на Windows Forms базирания GUI клиент са следните:

1. Разархивирайте архива, съдържащ Windows Forms приложението.
2. Отворете конфигурационния файл **`App.config`** и променете съдържанието на полето **`Path`**, така че да съдържа валиден адрес на уеб услугата, която преди малко инсталирахте.

Инсталиране на ASP.NET уеб приложението

Инсталирането на ASP. NET уеб приложението е подобно на инсталирането на уеб услугата и за това няма да го разглеждаме в детайли. Ще разгледаме само конфигурирането му:

1. След като инсталирате уеб приложението на системата за запознанства в Интернет, отворете конфигурационният файл **`Web.config`**.
2. Променете в него стойността на полето **`WebServiceGetPicturePage`** и полето **`DatingSiteASPNETClient.DatingSiteService.DatingSiteWebService`**, така че да сочат инсталираната преди това уеб услуга.

Използвана литература

1. Role-based Security with Forms Authentication By Heath Stewart – http://www.codeproject.com/aspnet/formsroleauth.asp
2. Maintaining Scroll Position on Postback By Steve Stchur – http://aspnet.4guysfromrolla.com/articles/111704-1.aspx
3. Enterprise Solution Patterns Using Microsoft .NET – http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpatterns/html/Esp.asp

4. CAPTCHA Image – http://www.brainjar.com/dotNet/CaptchaImage/
5. How to serve binary resources from a database in ASP.NET – http://weblogs.asp.net/cazzu/archive/2003/08/27/25568.aspx
6. Paging of Large Resultsets in ASP.NET By Jasmin Muharemovic – http://www.codeproject.com/aspnet/PagingLarge.asp
7. Unexpected Errors in Managed Applications – http://msdn.microsoft.com/msdnmag/issues/04/06/NET/default.aspx

Заключение към втория том

Авторският колектив, изготвил настоящата книга, силно се надява, че тя ви е дала полезни знания и умения за програмиране с .NET технологиите и ви е помогнала на професионалното развитие. Надяваме се, че не сме ви изгубили времето с големия обем информация.

За съжаление не можахме да покрием функционалността на .NET 2.0 платформата въпреки силното си желание и книгата си остана едно добро ръководство за .NET Framework 1.1. Извиняваме се на всички читатели за забавянето на втория том с почти една година. Надяваме се все пак да не е твърде късно той да бъде също така полезен, както и първия.

Ако имате въпроси или коментари, свързани с настоящата книга, отправяйте ги в нашия форум:

http://www.devbg.org/forum/index.php?showforum=30

Главният автор и ръководител на проекта, Светлин Наков, отправя покана към всички, които желаят да изпробват в практиката описаните в тази книга технологии и да се научат да ги прилагат в реални проекти, да се запишат за **БЕЗПЛАТНО** обучение в "Национална академия по разработка на софтуер":

http://academy.devbg.org/

Академията дава възможност за кратко време да овладеете съвременните софтуерни технологии, да придобиете практически умения за разработка на софтуер и да започнете успешно кариерата си на софтуерен инженер.

Надяваме се авторският колектив да намери сили за обновяването на книгата за .NET Framework 2.0 и следващи версии на платформата. Очакваме подкрепата на читателите.

Светлин Наков,
ноември, 2006

„Програмиране за .NET Framework" е уникално ръководство за платформата .NET. Въпреки, че не е учебник по програмиране, книгата е изключително подходяща както за начинаещия програмист, сблъскващ се за пръв път с .NET, така и за опитния разработчик на .NET приложения, целящ да систематизира и попълни знанията си.

Стоян Йорданов,
Software Design Engineer,
Microsoft Corp.

„Програмиране за .NET Framework" е първата чисто българска книга за Microsoft .NET технологиите. Тя представя на читателя в последователен, структуриран, достъпен и разбираем вид основните концепции за разработка на приложения с .NET Framework и езика C#. Книгата обхваща в детайли всички основни .NET технологии като набляга върху най-важните от тях: ADO.NET, ASP.NET, Windows Forms и XML уеб услуги.

Теодор Милев,
Управляващ директор на
"Майкрософт България"

Уеб сайт:
www.devbg.org/dotnetbook/

ISBN: 954-775-672-9
ISBN: 978-954-775-672-4

АВТОРИТЕ:

Александър Русев
Александър Хаджикръстев
Антон Андреев
Бранимир Ангелов
Васил Бакалов
Виктор Живков
Галин Илиев
Георги Пенчев
Деян Варчев
Димитър Бонев
Димитър Канев
Ивайло Димов
Ивайло Христов
Иван Митев
Лазар Кирчев
Манол Донев
Мартин Кулов
Михаил Стойнов
Моника Алексиева
Николай Недялков
Панайот Добриков
Преслав Наков
Радослав Иванов
Рослан Борисов
Светлин Наков
Стефан Добрев
Стефан Захариев
Стефан Кирязов
Стоян Дамов
Тодор Колев
Христо Дешев
Христо Радков
Цветелин Андреев
Явор Ташев