Светлин Наков и колектив

ПРОГРАМИРАНЕ ЗА .NET Framework

1 ТОМ

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

Кратко съдържание

Том 1

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
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, 2004-2005

Програмиране за .NET Framework

Официален сайт:

www.devbg.org/dotnetbook/

ISBN 954-775-505-6

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

Продукти

Телерик предлага компоненти за ускорена разработка на софтуерни приложения. Продуктите на Телерик имат за цел да улеснят изграждането и поддържането на портали, корпоративни уеб и интранет сайтове.

Телерик r.a.d.controls е най-пълният и иновативен набор от софтуерни компоненти за ASP.NET, позволяващ на професионалистите да разработват уеб приложения с функционалното богатство и бързина на традиционни десктоп приложения. r.a.d.controls се пред-лага във версии за Microsoft Sharepoint, Microsoft Content Management Server, DotNetNuke и ASP.NET 2.0.

Информация за свободни работни позиции и стажове в Телерик:
careers@telerik.com

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!

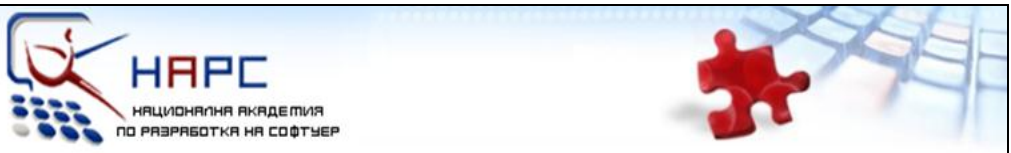
Учите безплатно във въведителните курсове и по стипендии от работодателите в следващите нива.

http://academy.devbg.org

Съдържание

Том 1

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

Предговор

Ако по принцип не четете уводите на книгите, помислете преди да пропуснете и този. Той е малко по-различен от всички останали, защото тази книга е също малко по-различна от всички останали.

Ако смятате, че ще ви досадим с общи приказки, можете да не се задълбочавате прекалено, но ви препоръчваме поне да преминете през следващите страници "по вентилаторната система", за да разберете какво ви предстои да научите от следващите страници. Ще разберете какво е .NET Framework, за какво служи, какви технологии обхваща и как настоящата книга от една идея се превърна в реалност.

Това е първата чисто българска книга за програмиране с .NET Framework и C#, но за сметка на това е една от най-полезните книги 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?

.NET Framework? Ама какво е това? Някаква нова измислица на Microsoft или просто поредния език за програмиране? Да не би да са направили нова версия на C++ или Java? А какъв е тоя език C#? Не мога ли да си пиша на C или C++? Какво е това среда за управлявано изпълнение на код? Не отмина ли вече времето на интерпретираните езици? Защо въобще трябва да сменяме добрите стари платформи с този .NET?

Ако нямате ясен отговор на всички тези въпроси, тази книга е за вас! Ако пък имате – тази книга също е за вас, защото едва ли знаете всичко за програмирането с .NET Framework и едва ли познавате добре всички по-важни технологии, свързани с него.

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.

Тази книга е за всички, които искат да се научат да програмират с .NET Framework и C#, както и за всички, които вече имат основни знания и умения в областта, но искат да ги разширят и да навлязат в някои от по-сложните технологии, с които нямат достатъчно опит.

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!

В нея няма да намерите обяснения за това какво е променлива, какво е тип данни, какво е условна конструкция, какво е цикъл и какво е функция. Очакваме читателят да е запознат добре с всички тези понятия и с основите на програмирането. Познанията по обектно-ориентирано програмиране (ООП) също ще са полезни, тъй като в книгата не се изясняват в дълбочина теоретичните концепции на ООП, а само средствата за тяхното прилагане в езика C#.

Какво представлява .NET Framework?

.NET Framework е съвременна платформа за разработка и изпълнение на приложения. Тя предоставя програмен модел, стандартна библиотека от класове и среда за контролирано изпълнение на програмен код.

.NET Framework поддържа различни езици за програмиране и позволява тяхната съвместна работа. .NET приложенията се пишат на езици от високо ниво (C#, VB.NET, Managed C++ и други) и се компилират до междинен език от ниско ниво, наречен IL (Intermediate Language). По време на изпълнение IL програмите (т. нар. управляван код) се компилират до инструкции за текущата хардуерна архитектура, съобразени с текущата операционна система, и след това се изпълняват от микропроцесора.

.NET Framework включва в себе си стандартна библиотека, която съдържа базова функционалност за разработка, необходима за повечето приложения, като вход/изход, връзка с бази данни, работа с XML, изграждане на уеб приложения, използване на уеб услуги, изграждане на графичен потребителски интерфейс и др.

Какво обхваща 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.

Настоящата книга обхваща всички тези концепции и технологии, свързани с разработката на приложения за .NET Framework. Тя има за цел да запознае читателя с принципите на разработка на приложения за Microsoft

.NET Framework и да даде широки познания по всички по-важни технологии, свързани с него.

Най-важните теми, които ще бъдат разгледани, са: архитектура на .NET Framework, управлявана среда за изпълнение на код (CLR), езикът C# и реализация на обектно-ориентирано програмиране с неговите средства, обща система от типове (CTS), основна библиотека от класове (Framework Class Library), достъп до бази от данни с ADO.NET, работа с XML, създаване на графичен потребителски интерфейс с Windows Forms и уеб-базирани приложения с ASP.NET. Ще бъде обърнато внимание и на 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. Не се обръща много внимание на новостите в .NET Framework 2.0, Visual Studio 2005 и SQL Server 2005, тъй като по време на разработката на книгата тези продукти и технологии все още не бяха официално излезли на пазара и тяхното бъдеще не беше съвсем ясно.

Въпреки предстоящото излизане на .NET Framework 2.0, настоящата книга си остава изключително полезна, тъй като в същината си версия 2.0 не носи фундаментални промени, а по-скоро разширява вече съществуващите технологии, които ще разгледаме в книгата.

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.

Защо C#?

Всички примери в книгата са написани на езика C#, въпреки, че .NET Framework поддържа много други езици. Този избор е направен по няколко причини:

- C# е препоръчваният език за програмиране за .NET Framework. Архитектите на езика специално са го проектирали за .NET Framework и са го съобразили с особеностите на платформата още по време на дизайна. C# наследява простотата на Java, мощността на C++ и силните черти на Delphi. Той притежава максимално стегнат и ясен синтаксис.
- В България C# е най-популярният от .NET езиците и се използва най-масово в българските софтуерни компании.
- C# е от семейството на C-базираните езици и синтактично много прилича на Java, C++, C и PHP. Много хора, които не знаят езика, биха разбрали примерите без особени усилия.
- За C# има повече статии в специализираните сайтове и лични дневници (blogs) в Интернет. Общността на C# разработчиците е по-добре развита, отколкото на разработчиците на другите .NET езици.
- Поради голямата популярност на езика C# за него има по-добра поддръжка от инструментите за разработка.
- Езици като C++, Visual Basic и JScript не са проектирани специално за .NET Framework, а са адаптирани допълнително към него чрез редица изменения и добавки. В следствие на това те запазват някои синтактични особености, които не са удобни при работата с .NET.

Ако сега започвате да изучавате .NET Framework, Ви препоръчваме да стартирате от езика C#. След като го овладеете, можете да опитате и другите .NET езици, но за начало C# е най-подходящ.

По принцип езикът C++ може да се използва при програмиране с .NET Framework, но това се препоръчва само при някои много специфични приложения. Този език по първоначален замисъл не е проектиран за .NET платформата и има съвсем друго предназначение. Той е много по-сложен и труден от C# и затова е по-добре да използвате C#, дори ако трябва да го учите от начало. Ако вече знаете C++, няма да ви е трудно да овладеете C# и когато го направите, ще се убедите, че с него се работи много по-лесно.

Въпреки, че езикът Visual Basic .NET (VB.NET) има някои предимства и се използва масово по света, за предпочитане е да ползвате C# при изграж-

дане на .NET приложения. Езикът Visual Basic е масово разпространен по исторически причини (благодарение най-вече на Бил Гейтс). Някои специалисти изказват силно негативни мнения срещу BASIC и произлизащите от него езици, докато други (включително и Microsoft) го подкрепят и препоръчват.

Ще си позволим да цитираме изказването на един от най-известните учени в областта на компютърните науки проф. д-р Едсгар Дейкстра за езика BASIC, от който произлиза VB.NET:

Практически е невъзможно да научиш на добро програмиране студенти, които са имали предишен досег до езика BASIC – като потенциални програмисти, те са мисловно осакатени, без надежда за възстановяване.

Едсгар Дейкстра

Горният цитат се отнася за старите версии на езика BASIC. VB.NET е вече съвременен обектно-ориентиран език, който не отстъпва по нищо на C#, освен че има малко по-нетрадиционен синтаксис (в сравнение със семейството на C-базираните езици).

.NET Framework позволява всеки да програмира на любимия си език. Изборът си е лично ваш. Ние можем само да ви дадем препоръки. За целите на настоящата книга авторският колектив е избрал езика C# и препоръчва на читателите да започнат от него.

Поглед към съдържанието на книгата

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.

Глава 1. Архитектура на .NET Framework

В глава 1 е представена платформата .NET, която въплъщава визията на Microsoft за развитието на информационните и софтуерните технологии, след което е разгледана средата за разработка и изпълнение на .NET приложения Microsoft .NET Framework.

Обръща се внимание на управлявания код, на езика IL, на общата среда за контролирано изпълнение на управляван код (Common Language Runtime) и на модела на компилация и изпълнение на .NET кода. Разглеждат се още Common Language Specification (CLS), Common Type System (CTS), Common Language Infrastructure (CLI), интеграцията на различни езици, библиотеката от класове Framework Class Library и интегрираната среда за разработка Visual Studio .NET.

Автори на главата са Виктор Живков и Николай Недялков. Текстът е написан с широко използване на лекциите на Светлин Наков по темата и е редактиран от Иван Митев и Светлин Наков.

Глава 2. Въведение в езика C#

Глава 2 разглежда езика C#, неговия синтаксис и основни концепции. Представя се средата за разработка Visual Studio .NET 2003 и се демонстрира работата с нейния дебъгер. Отделя се внимание на типовете данни, изразите, програмните конструкции и конструкциите за управление в езика C#. Накрая се демонстрира колко лесно и полезно е XML документирането на кода в C#.

Автор на главата е Моника Алексиева. Текстът е базиран на лекцията на Светлин Наков по същата тема и е редактиран от Панайот Добриков и Преслав Наков.

Глава 3. Обектно-ориентирано програмиране в .NET

В глава 3 се прави кратък обзор на основните принципи на обектно-ориентираното програмиране (ООП) и средствата за използването им в .NET Framework и езика C#. Представят се типовете "клас", "структура" и "интерфейс" в C#. Въвежда се понятието "член на тип" и се разглеждат видовете членове (член-променливи, методи, конструктори, свойства, индексатори и др.) и тяхната употреба. Разглежда се наследяването на типове в различните му аспекти и приложения. Обръща се внимание и на полиморфизма в C# и свързаните с него понятия и програмни техники. Накрая се дискутират някои утвърдени практики при създаването на ефективни йерархии от типове.

Автор на главата е Стефан Кирязов. Текстът е написан с широко използване на лекции на Светлин Наков и е редактиран от Цветелин Андреев и Панайот Добриков.

Глава 4. Обработка на изключения в .NET

В глава 4 се разглеждат изключенията в .NET Framework като утвърден механизъм за управление на грешки и непредвидени ситуации. Дават се обяснения как се прихващат и обработват изключения. Разглеждат се начините за тяхното предизвикване и различните видове изключения в .NET Framework. Дават се примери за дефиниране на собствени (потребителски) изключения.

Автори на главата са Явор Ташев и Светлин Наков. Текстът е написан с широко използване на лекции на Светлин Наков по темата. Редактор е Мартин Кулов.

Глава 5. Обща система от типове

В глава 5 се разглежда общата система от типове (Common Type System) в .NET Framework. Обръща се внимание на разликата между стойностни и референтни типове, разглежда се основополагащият тип **`System.Object`** и йерархията на типовете, произлизаща от него. Дискутират се и някои особености при работа с типове – преобразуване към друг тип, проверка на тип, клониране, опаковане, разопаковане и др.

Автор на главата е Светлин Наков. Текстът е базиран изцяло на лекцията на Светлин Наков по същата тема и е редактиран от Преслав Наков и Панайот Добриков.

Глава 6. Делегати и събития

В глава 6 се разглежда референтният тип "делегат". Илюстрирани се начините за неговото използване, различните видове делегати, както и негови характерни приложения. Представя се понятието "събитие" и се обяснява връзката му с делегатите. Прави се сравнение между делегатите и интерфейсите и се дават препоръки в кои случаи да се използват едните и в кои – другите.

Автор на главата е Лазар Кирчев. Текстът е базиран на лекцията на Светлин Наков по същата тема.

Глава 7. Атрибути

В глава 7 се разглежда какво представляват атрибутите в .NET Framework, как се прилагат и къде се използват. Дават се обяснения как можем да дефинираме собствени атрибути и да извличаме приложените атрибути от метаданните на асемблитата.

Автори на главата са Преслав Наков и Панайот Добриков. Текстът е базиран основно на лекцията на Светлин Наков по същата тема и е редактиран от него.

Глава 8. Масиви и колекции

В глава 8 се представят масивите и колекциите в .NET Framework. Разглеждат се видовете масиви – едномерни, многомерни и масиви от масиви (т. нар. назъбени масиви), както и базовият за всички масиви тип **`System.Array`**. Дискутират се начините за сортиране на масиви и търсене в тях. Разглеждат се колекциите и тяхната реализация в .NET Framework, класовете **`ArrayList`**, **`Queue`**, **`Stack`**, **`Hashtable`** и **`SortedList`**, както и интерфейсите, които те имплементират.

Автори на главата са Стефан Добрев и Деян Варчев. Текстът е базиран на лекцията на Светлин Наков по същата тема и е редактиран от него.

Глава 9. Символни низове

В глава 9 се разглежда начинът на представяне на символните низове в .NET Framework и методите за работа с тях. Обръща се внимание на кодиращите схеми, които се използват при съхраняване и пренос на текстова информация. Разглеждат се подробно различните начини за манипулиране на низове, както и някои практически съображения при работата с тях. Демонстрира се как настройките за държава и регион (култура) определят вида на текста, показван на потребителите, и как можем да форматираме изхода в четлив и приемлив вид. Разглеждат се също и начините за преобразуване на вход от потребителя от текст в обект от стандартен тип, с който можем лесно да работим.

Автори на главата са Васил Бакалов и Александър Хаджикръстев. В текста е широко използвана лекцията на Светлин Наков по същата тема. Главата е редактирана от Иван Митев.

Глава 10. Регулярни изрази

В глава 10 се разглеждат регулярните изрази, набиращи все по-голяма популярност сред разработчиците на софтуер при решаването на проблеми, свързани с обработката на текст. Дискутират се произходът и същността на регулярните изрази, техният синтаксис и основните правила при конструирането им. В главата е предложено кратко представяне на основните дейности, при които е подходящо използването на регулярни изрази, и са дадени конкретни насоки как можем да правим това със средствата на .NET Framework. Разглежда се инструментариумът, за работа с регулярни изрази, който стандартната библиотека с класове предоставя, и се описват най-важните методи, съпроводени с достатъчно примери.

Автор на главата е Георги Пенчев. При изготвянето на текста е частично използвана лекцията на Светлин Наков по темата. Технически редактор е Иван Митев.

Глава 11. Вход/изход

В глава 11 се разглежда начинът, по който се осъществяват вход и изход от дадена програма в .NET Framework. Представят се различните видове потоци – абстракцията, която позволява връзката на програмата с някакво устройство за съхранение на данни. Обяснява се работата на четците и писачите, които обвиват потоците и така улесняват тяхното използване. Накрая, се прави преглед на средствата, които .NET Framework предоставя за работа с файлове и директории и за наблюдение на файловата система.

Автор на главата е Александър Русев. Текстът е базиран на лекцията на Светлин Наков по същата тема и е редактиран от Галин Илиев и Светлин Наков.

Глава 12. Работа с XML

В глава 12 се разглежда работата с XML в .NET Framework. Обяснява се накратко какво представлява езикът XML. Обръща се внимание на приликите и разликите между него и HTML. Разглеждат се приложенията на XML, пространствата от имена и различните схеми за валидация на XML документи (DTD, XSD, XDR). Представят се средствата на Visual Studio .NET за работа с XSD схеми. Разглеждат се особеностите на класическите XML парсери (DOM и SAX) и как те са имплементирани в .NET Framework. Описват се подробно класовете за работа с DOM парсера (**`XmlNode`** и **`XmlDocument`**) и ролята на класа **`XmlReader`** при SAX парсерите в .NET Framework. Обръща се внимание на начина на работа на класа **`XmlWriter`** за създаване на XML документи. Дискутират се начините за валидация на XML документи спрямо дадена схема. Разглежда се поддръжката в .NET Framework и на някои други XML-базирани технологии като XPath и XSLT.

Автор на главата е Манол Донев, а редактори са Иван Митев и Светлин Наков. Текстът широко използва лекцията на Светлин Наков по същата тема.

Глава 13. Релационни бази от данни и MS SQL Server

В глава 13 се разглеждат системите за управление на релационни бази от данни. Обясняват се свързаните с тях понятия като таблици, връзки, релационна схема, нормализация, изгледи, ограничения, транзакции, съхранени процедури и тригери. Прави се кратък преглед на езика SQL, използван за манипулиране на релационни бази от данни.

След въведението в проблематиката на релационните бази от данни се прави кратък преглед на Microsoft SQL Server, като типичен представител на RDBMS сървърите. Разглеждат се неговите основни компоненти и инструменти за управление. Представя се използваното от него разширение на езика SQL, наречено T-SQL, и се дискутират основните DDL, DML и DBCC команди. Обръща се внимание на съхранените процедури в SQL Server и се обяснява как той поддържа някои важни характеристики на една релационна база от данни, като транзакции, нива на изолация и др.

Автор на главата е Стефан Захариев. В текста са използвани учебни материали от Бранимир Гюров, Светлин Наков и Стефан Захариев. Редактор е Светлин Наков.

Глава 14. ADO.NET и работа с данни

В глава 14 се разгледат подробно двата модела за достъп до данни, реализирани в ADO.NET – свързан и несвързан. Описва се програмният модел на ADO.NET, неговите компоненти и доставчиците на данни. Обяснява се кои класове се използват за свързан достъп до данни, и кои – за несвързан.

При разглеждането на свързания модел за достъп до данни се обръща внимание на доставчика на данни **SqlClient** за връзка с MS SQL Server и се обяснява как се използват класовете **`SqlConnection`**, **`SqlCommand`** и **`SqlDataReader`**. Разглежда се работата с параметризирани заявки и използването на транзакции от ADO.NET. Дава се пример за достъп и до други бази от данни през OLE DB. Разглеждат се и някои проблеми при работа с дати и съхранение на графични изображения в базата данни.

При разглеждането на несвързания модел за достъп до данни се дискутират в детайли основните ADO.NET класове за неговата реализация – **`DataSet`** и **`DataTable`**. Дават се примери и обяснения как се използват ограничения, изрази, релации и изгледи в обектния модел **`DataSet`**. Обръща се специално внимание на класа **`DataAdapter`** и вариантите за неговото използване при зареждане на данни и обновяване на базата от данни. Разглеждат се подходите за решаване на конфликти при нанасяне на промени в базата данни. Дискутират се и начините за връзка между ADO.NET и XML, а накрая се разглеждат проблемите със сигурността в приложенията, използващи бази от данни.

Автори на главата са Христо Радков (частта за свързания модел) и Лазар Кирчев (частта за несвързания модел). Главата е разработена с широко използване на лекцията на Бранимир Гюров и Светлин Наков по същата тема. Редактори са Светлин Наков и Мартин Кулов.

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.

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 приложения. 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 визуа-

лизират 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.

Автор на главата е Михаил Стойнов. Текстът е базиран на лекцията на Михаил Стойнов по същата тема.

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.

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.

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 асембли. Демонстрира се как може да се извлече информация за типовете в дадено асембли 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 асембли, 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 по-обстойно 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 й 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. Технически редактор е Светлин Наков.

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 Светлин Наков.

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. Коректор е Соня Бибиликова.

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. Редактор е Светлин Наков.

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. Редактор на текста е Светлин Наков.

За използваната терминология

Тъй като настоящият текст е на български език, ще се опитаме да ограничим употребата на английски термини, доколкото е възможно. Съществуват обаче три основателни причини да използваме и английските термини наред с българските им еквиваленти:

- По-голямата част от техническата документация за .NET Framework е на английски език (повечето книги и в частност MSDN Library) и затова е много важно читателите да знаят английския еквивалент на всеки използван термин.
- Много от използваните термини не са пряко свързани с .NET и са навлезли отдавна в програмисткия жаргон от английски език (например "дебъгвам", "компилирам" и "плъгин"). Тези термини ще бъдат изписвани най-често на кирилица.
- Някои термини (например "framework" и "deployment") са трудно преводими и трябва да се използват заедно с оригинала в скобки. В настоящата книга на места такива термини са превеждани по различни начини (според контекста), но винаги при първо срещане се дава и оригиналният термин на английски език.

Конвенция за кода

С цел уеднаквяване на стила на кода във всички примери от книгата, в примерите и демонстрациите от лекциите, както и в практическия проект, е въведена конвенция за кода, която включва редица препоръки за форматирането на кода, имената на типове, членове и променливи, елементи от потребителския интерфейс и други. Ще обясним по-важните от тях:

Константите пишем с главни букви

Примери:

```
private const int MAX_VALUE = 4096;
private const string INPUT_FILE_NAME = "input.xml";
```

Това е утвърдена практика, възприета от повечето програмисти на C, C++, Java и C#.

Член-променливите пишем с префикс "m"

Примери:

```
private Hashtable mUsersProfiles;
private ArrayList mUsers;
```

Тази конвенция не е стандартна, но тъй като Microsoft нямат официална препоръка по този въпрос, ние възприехме тази конвенция за именуване на член-променливите, за да ги отличаваме от останалите променливи. Префиксът "**m**" произхожда от думата "**member**" (член).

Параметрите на методите пишем с префикс "a"

Пример:

```
public void IsLoginValid(string aUserName, string aPassword)
{
   // ...
}
```

Тази конвенция също не е стандартна, но ние я възприехме, за да можем лесно да отличаваме параметрите в методите от останалите променливи, което често пъти е много полезно. Префиксът "**a**" произхожда от думата "**argument**" (аргумент на метод).

Именуване на идентификатори

Възприели сме конвенция за именуване на идентификаторите, която е близка до официалните препоръки на Microsoft (за случаите, в които Microsoft са дали препоръки) и е съобразена с принципите за именуване на член-променливи и параметри, които вече разгледахме. Ето как изглежда тази конвенция:

Идентификатор	Стил	Пример
пространство от имена (namespace)	Pascal Case	**System.Windows.Forms**
тип (клас, структура, ...)	Pascal Case	**TextWriter**
интерфейс (interface)	Pascal Case, префикс "**I**"	**ISerializable**
изброен тип (enum type)	Pascal Case	**FormBorderStyle**
изброена стойност (enum value)	Pascal Case	**FixedSingle**
поле само за четене (read-only field)	Pascal Case	**UserIcons**
поле-константа (constant)	UPPERCASE	**MAX_VALUE**
свойство (property)	Pascal Case	**BorderColor**

събитие (event)	Pascal Case	`SizeChanged`
метод (method)	Pascal Case	`ToString()`
член-променлива (field)	Pascal Case, префикс "`m`"	`mUserProfiles`
статична член-променлива (static field)	Pascal Case, префикс "`m`"	`mTotalUsersCount`
параметър на метод (parameter)	Pascal Case, префикс "`a`"	`aFileName`
локална променлива (local variable)	Camel Case	`currentIndex`

Именуване на контроли

При именуване на контроли използваме Pascal Case и представка, която съответства на техния тип. Не слагаме префикс "`m`", когато контролата е член-променлива:

Контрола	Пример
`Button`	`ButtonOk`, `ButtonCancel`
`Label`	`LabelCustomerName`
`TextBox`	`TextBoxCustomerName`
`Panel`	`PanelCustomerInfo`
`Image`	`ImageProduct`

Конвенции за базата данни

Използваме множествено число за именуване на таблици (например `Users`, `Countries`, `StudentsCourses`, …). При имената на колоните в таблица използваме Pascal Case (например `UserName`, `MessageSender`, `UserId` и т.н.).

Служебните думи в езика SQL (например `SELECT`, `CREATE TABLE`, `FROM`, `INTO`, `ORDER BY` и др.) изписваме с главни букви.

Как възникна тази книга?

Историята на тази книга е дълга и интересна.

Няколко години след официалното излизане на .NET платформата, през 2002 г. .NET Framework вече беше навлязъл широко на пазара и много български фирми разработваха .NET приложения. Езикът C# и .NET платформата вече бяха добре познати сред софтуерните специалисти, но по университетите все още никой не преподаваше тези технологии.

В този момент в Софийски университет възникна курсът "Програмиране за платформа .NET".

Курсът по програмиране за платформа .NET в СУ (2002/2003 г.)

Курсът "Програмиране за платформа .NET" в Софийски университет беше организиран през летния семестър на учебната 2002/2003 г. от група студенти с изявен интерес към .NET технологиите, някои от които имаха вече натрупан сериозен практически опит като .NET разработчици.

Преподавателският екип беше в състав Светлин Наков (работещ тогава в Мусала Софт), Стоян Йорданов (работещ тогава в Рила Солюшънс), Георги Иванов (работещ тогава във WebMessenger) и Николай Недялков (работещ тогава в Информационно обслужване).

Курсът (http://www.nakov.com/dotnet/2003/) обхващаше всички основни технологии, свързани с .NET Framework. Интересът към него беше много голям. Над 300 студента преминаха обучението, което беше с обем 60 учебни часа. Много от тях след това започнаха професионалната си кариера като .NET програмисти.

По време на семестъра бяха разработени авторски учебни материали за повечето от темите, които по-късно бяха използвани при изготвянето на лекции по "Програмиране за .NET Framework", на които е базирана настоящата книга.

Проектът на Microsoft Research и БАРС

Две години по-късно Microsoft Research отправиха предложение към Софийски университет за участие в академичен проект за създаване на учебно съдържание и учебни материали по дисциплини, изучаването на които е базирано на технологиите на Microsoft.

Екипът на Светлин Наков, съвместно с Българска асоциация на разработчиците на софтуер и Софийски университет предложиха проект за разработка на изчерпателно учебно съдържание и провеждане на университетски курсове по "Програмиране за .NET Framework". Проектът беше одобрен и частично финансиран от Microsoft Research.

Така започна съставянето на учебните материали, върху които е базирана настоящата книга. За година и половина бяха изработени повече от 2000 PowerPoint слайда по 26 теми, съдържащи над 600 примера, около 200 демонстрации на живо и над 300 задачи за упражнения. Учебните материали са с много високо качество и предоставят задълбочена информация по всички по-важни технологии, свързани с програмирането с .NET Framework. По някои от темите лекциите се получиха значително по-добри от официалните учебни материали на Microsoft (т. нар. Microsoft Official Curriculum). Лекциите са достъпни за свободно изтегляне от сайта на книгата.

Курсът по програмиране за .NET Framework в СУ (2004/2005 г.)

По разработените вече учебни материали през зимния семестър на 2004/2005 г. беше проведен курс във Факултета по математика и информатика на Софийски университет с продължителност 90 учебни часа.

Курсът (http://www.nakov.com/dotnet/) беше организиран от Светлин Наков и неговия екип – Бранимир Гюров, Мартин Кулов, Георги Иванов, Михаил Стойнов и Ивайло Христов. Интересът към курса отново беше голям и стотици студенти избраха да преминат обучението. Мнозина от тях след това започнаха работа като .NET програмисти във водещи български софтуерни компании.

Няколко месеца след приключване на курса започна писането и на настоящата книга по материалите, използвани в лекциите.

Курсът по програмиране за .NET Framework в СУ (2005/2006 г.)

През зимния семестър на 2005/2006 г. във Факултета по математика и информатика на Софийски университет "Св. Климент Охридски" отново се организира курс по .NET Framework (http://www.devbg.org/dotnetcourse/) с продължителност 90 учебни часа.

Преподавателският екип е съставен от представители на авторския колектив, разработил настоящата книга: Светлин Наков, Ивайло Христов, Михаил Стойнов, Галин Илиев, Васил Бакалов, Стефан Захариев, Радослав Иванов, Антон Андреев, Стефан Кирязов и Виктор Живков.

Курсът се провежда по официалните лекции и учебни материали, разработени по съвместния проект между Microsoft Research, Софийски университет и БАРС, които са достъпни за свободно изтегляне от сайта на курса.

Настоящата книга се използва като официален учебник в курса.

Проектът за настоящата книга

Първоначално идеята беше да се разпишат като текст изготвените вече лекции и да се компилира учебник за курсовете по програмиране за .NET Framework. По-късно проектът силно се разрасна и в него се включиха над 30 души. Появиха се допълнителни теми, появиха се и множество допълнения към обхванатите в лекциите теми.

Книгата е безплатна!

Настоящата книга се разпространява напълно безплатно в електронен вид по лиценз, който позволява използването ѝ за всякакви цели, включително и в комерсиални проекти. Книгата се разпространява и в хартиен вид срещу заплащане, което покрива разходите по отпечатването и разпространението ѝ, без да се реализира печалба.

Екипът, реализирал идеята

Екипът, написал настоящата книга, е съставен от хора, които имат силен интерес към .NET технологиите и желаят безвъзмездно да споделят своя опит като участват в написването на една или няколко от темите. Някои от участниците в екипа са бивши студенти, посещавали курсовете по .NET Framework в Софийски университет, други са членове на Софийската .NET потребителска група (www.sofiadev.org), а трети – разработчици, които от някъде са научили за проекта. Всички автори, съавтори и редактори от екипа по разработката на книгата са програмисти с реален практически опит.

Участниците в проекта дадоха своя труд безвъзмездно, без да получат материални или други облаги, защото съзнаваха липсата на добра книга за .NET Framework на български език и имаха силно желание да помогнат на своите настоящи и бъдещи колеги да навлязат с много по-малко усилия в .NET технологиите.

Процесът на работа

Написването на книгата отне около 6 месеца. Екипът беше ръководен от Светлин Наков, който има богат опит с писането на статии, презентации и книги и притежава добри технически познания по .NET Framework. Екипът се събираше на всеки 2 седмици за да дискутира напредъка по задачите и проблемите, възникнали по време на работата по проекта.

Работата по всяка тема изискваше нейният автор да предава по 10-15 страници на всеки 2 седмици. Този подход доведе до намаляване на риска от закъснение на работата по темите, и позволи проблемите да бъдат идентифицирани и решавани още при възникването им. В крайна сметка проектът завърши успешно, макар и доста след планираните първоначално срокове.

По време на работата възникваха проблеми, породени от голямото натоварване на авторите на работното им място. Някои автори трудно успяваха да спазят обещаните срокове (а други дори никога не са ги спазвали). По време на поправителната сесия някои студенти имаха сериозни трудности. Въпреки това само един участник, който се включи в проекта, в последствие се отказа. Всички останали написаха успешно своите теми.

За улесняване на съвместната работа бе използвана системата за екипна работа по проекти, предлагана свободно от портала sciforge.org. За целите на книгата в SciForge беше регистриран и използван проект "Книга за .NET Framework", който все още е публично достъпен от адрес http://sciforge.org/projects/dotnetbook/. Беше използвана системата за контрол на версиите Subversion, форумът и пощенският списък (mailing list), предлагани от SciForge.

За да се уеднаквят стиловете и форматирането във всички глави, беше разработено специално "ръководство за писателите", което дефинираше строги правила, свързани със стила на изказ, структурирането на текста, форматирането на кода, примерите, таблиците, схемите, картинките и т.н.

Бяха разработени конвенции за кода, речник на преводните думи и други полезни стандарти. За всяка глава беше направен шаблон за MS Word 2003, в който авторите трябваше да пишат. Всички тези усилия силно ограничиха различията в стила и форматирането между отделните глави на книгата.

Всяка тема, след написването ѝ, беше редактирана и редактирана от поне един редактор. Първоначално всички редакции и рецензии се извършваха от ръководителя на проекта Светлин Наков, но по-късно към редактирането се присъединиха и други участници. В резултат на общите усилия съдържанието на всички теми е на добро техническо ниво и добре издържано откъм стил.

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 JCI (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 TnDSoft (www.tndsoft.com). 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-ориенти-

рано 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 изпълнителен директор на фирма КодАтест (www.codeattest.com), в която разработва системи за управление на качеството и автоматизация на софтуерното производство. Той има дългогодишен професионален опит като разработчик и ръководител в различни по големина проекти за частния и обществения сектор. Интересите му са в областта на продуктите и технологиите на Microsoft. Мартин е сертифициран от Microsoft разработчик по програмите MCSD и MCSD.NET (Charter Member). Той е магистър инженер при Факултета по комуникационна техника и технологии на Технически университет – София. През 2004 г. той участва като лектор в курсовете "Програмиране за .NET Framework" и "Качествен програмен код" в Софийски университет "Св. Климент Охридски". Мартин е лектор и на семинари на Microsoft, свързани с .NET технологиите и разработката на софтуер. Той е почетен член на Българската асоциация на разработчиците на софтуер и член на SofiaDev .NET потребителската група. Можете да се свържете с него по e-mail: martin@codeattest.com или чрез неговия личен уеблог: 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 информаци-

онните 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 по-късно 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.

Светлин Наков

Светлин Наков е директор на Националната академия по разработка на софтуер (http://academy.devbg.org), където обучава софтуерни специалисти за практическа работа в ИТ индустрията. 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" и "Качествен програмен код". 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" и "Java за цифрово подписване на документи в уеб". 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 Verix (www.verix.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: skiryazov@verix.bg.

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 **t**elerik (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". 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 и няколко по-малки информационни системи, динамични Интернет сайтове 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 разработчик във фирма Komero Technologies (www.komero.net). Той отговаря основно за UNIX базираните решения и за модули, свързани с вътрешния процес на разработка. В момента Цветелин е студент във Факултета по математика и информатика на Софийски университет "Св. Климент Охридски" и е професионално сертифициран от Sun. Неговите интереси са основно в областта на Java и UNIX технологиите, но обхващат и области от .NET платформата, изкуствен интелект, мрежова сигурност, анализ на изисквания, софтуерни архитектури и дизайн. Личният сайт на Цветелин е достъпен от адрес: www.flowerlin.net.

Явор Ташев

Явор Ташев е софтуерен разработчик във фирма TND Soft (www.tndsoft.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 и комуникационни системи, базирани на 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 доброволни 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.

Благодарностите са отправени към ст. ас. Елиза Стефанова (която оформи изключително убедително текста на предложението за проекта към Microsoft Research) и доц. Магдалина Тодорова (която пое ролята на административен ръководител при взаимоотношенията с Microsoft).

По-късно, когато проектът на MS Research приключи и започна работата по настоящата книга, ФМИ предостави зали и техника за провеждане на регулярните срещи на авторския колектив.

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.

Други

Изказваме благодарности още към:

- Георги Иванов, ръководител на проекти във фирма Sciant (www.sciant.com), участник в преподавателския екип на курсовете по "Програмиране за .NET Framework". Участник в създаването на лекциите, по които е изградена настоящата книга.
- Стоян Йорданов, софтуерен инженер в Microsoft Corporation, Redmond (www.microsoft.com), участник в преподавателския екип на курсовете по "Програмиране за .NET Framework". Участник в създаването на лекциите, по които е изградена настоящата книга.
- Бранимир Гюров, частичен съавтор на една от главите на книгата, участник в преподавателския екип на курса "Програмиране за .NET Framework". Участник в създаването на лекциите, на които се основава настоящата книга.

- Невена Партинова, графичен дизайнер. Благодарности за изготвянето на корицата на книгата и за цялото търпение по време на продължителните дискусии за графичния дизайн и цветовата гама.
- Михаил Балабанов, преводач и автор на спецификации за превод на софтуер, участник в превода на OpenOffice.org. Благодарности за помощта при превода на някои технически термини.
- Никола Касев, взел участие при създаването на лекциите, по които е изградена настоящата книга.
- Свилена Момова, частичен съавтор на една от главите на книгата.
- Веселин Райчев, частичен съавтор на една от главите на книгата.

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,
24.09.2005 г.

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 1. Архитектура на платформата .NET и .NET Framework

Необходими знания

- Познания по програмиране
- Езици за програмиране
- Среди за разработка на софтуер

Съдържание

- Какво е .NET?
- Архитектура на платформата Microsoft .NET
- Какво е .NET Framework?
- Архитектура на .NET Framework
- Common Language Runtime (CLR)
- Управляван код
- Междинен език IL
- Модел за изпълнение на IL кода
- Асемблита и метаданни
- .NET приложения
- Домейни на приложението
- Common Language Specification (CLS), Common Type System (CTS)
- Common Language Infrastructure (CLI) и интеграцията на различни езици
- Framework Class Library
- Интегрирана среда за разработка Visual Studio .NET

В тази тема ...

В настоящата тема ще представим платформата .NET, която въплъщава визията на Microsoft за развитието на информационните и софтуерните технологии, след което ще разгледаме средата за разработка и изпълнение на .NET приложения Microsoft .NET Framework. Ще обърнем внимание на управлявания код, на езика IL, на общата среда за контролирано

изпълнение на управляван код (Common Lnaguage Runtime) и на модела на компилация и изпълнение на .NET кода. Ще разгледаме още Common Language Specification (CLS), Common Type System (CTS), Common Language Infrastructure (CLI), интеграцията на различни езици, библиотеката от класове Framework Class Library и интегрираната среда за разработка Visual Studio .NET.

Какво представлява платформата .NET?

Microsoft дефинират платформата .NET като съвкупност от технологии, които свързват хората с информацията – навсякъде, по всяко време, от всяко устройство. Това определение звучи като маркетингова пропаганда, но .NET е не само технология, тя е и идеология. Платформата въплъщава визията на Microsoft, че информацията трябва да бъде максимално достъпна за хората.

.NET платформата осигурява стандартизирана инфраструктура за разработка, използване, хостинг и интеграция на .NET приложения и XML уеб услуги, базирана на .NET сървърите на Microsoft, средствата за разработка (.NET Framework и Visual Studio .NET), идеологията на smart клиентите и т. нар. .NET Building Block Services.

Визията на Microsoft

Визията на Microsoft за .NET е да създадат платформа, която да може да обединява хетерогенна инфраструктура от сървъри, да интегрира бизнес процесите на различни компании по стандартен начин, и да предоставя на потребителите достъп до информацията, която им е нужна, по всяко време, от всяко място и от всяко устройство. Както ще видим по-нататък, Microsoft са направили голяма крачка напред към реализирането на тази визия, като са поставили една стабилна технологична основа за разработка и изпълнение на приложения – Microsoft .NET Framework.

Разграничавайте понятията "платформа .NET" и ".NET Framework"!

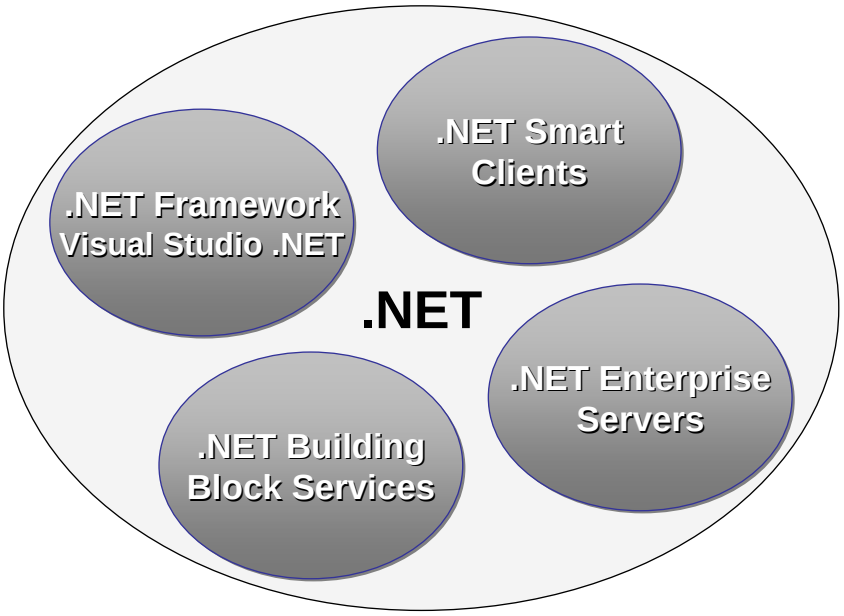
.NET платформата е визията на Microsoft за развитието на технологиите и осигурява глобална инфраструктура за реализацията на тази визия.

.NET Framework е само част от .NET платформата – тази част, която е насочена към разработчиците на софтуер. Тя осигурява среда за разработка и контролирано изпълнение на .NET приложения и предоставя програмен модел и библиотеки от класове за разработка, независима от езиците за програмиране.

Имайте предвид, че много често под .NET се подразбира не платформата .NET, а средата .NET Framework, например ".NET език", ".NET приложение" и т. н. В настоящата книга също ще подразбираме под .NET не .NET платформата, а .NET Framework.

Архитектура на .NET платформата

Платформата .NET обединява в себе си четири технологични и идеологически компонента: инфраструктурата от сървъри .NET Enterprise Servers, средствата за разработка .NET Framework и Visual Studio .NET 2003, глобалните услуги .NET Building Block Services и идеологията .NET Smart Clients:



Всеки един от изброените компоненти на .NET платформата е достатъчно обемна тема, за да й се посвети цяла отделна книга, но нашата цел е само да се запознаем накратко с посочените технологии и идеологии, без да навлизаме в подробности. Нека сега ги разгледаме една по една.

.NET Enterprise Servers

.NET Enterprise Servers предоставят сървърната инфраструктура на .NET платформата и същевременно среда за изпълнение, управление и интеграция на XML уеб услуги.

Ключови характеристики

Ключовите характеристики на .NET Enterprise сървърите са:

- Силна поддръжка на XML – всички .NET сървъри използват широко XML стандарта за представяне и обмяна на информация.
- Висока надеждност – ключова характеристика, изключително важна за бизнеса.
- Добра скалируемост – възможност за поемане на огромно натоварване при необходимост.

- Оркестрация на бизнес процесите в приложенията и услугите (business process orchestration) – дава се възможност за схематично дефиниране на работните процеси по утвърдени стандарти (като BPEL) и контролираното им изпълнение, наблюдение и управление.
- Повишена сигурност – сигурността е основна архитектурна концепция при .NET сървърите.
- Лесно управление – лесната за администриране, настройка, наблюдение и управление на работата на сървърите.

По-важните сървърни продукти

Microsoft разработват сървърни продукти от много години и в момента предлагат цяло семейство от специализирани сървъри, насочени към различни бизнес нужди. Ще дадем съвсем кратко описание на най-важните от тях:

- Microsoft Windows Servers Family – представлява фамилия сървърни операционни системи (като Windows 2000 Server и Windows 2003 Server).
- Microsoft Internet Information Server – представлява уеб сървър, който е част от Windows. Служи за хостинг на уеб сайтове със статично и динамично съдържание.
- Microsoft SQL Server – служи за управление на релационни бази от данни, многомерни данни и XML.
- Microsoft BizTalk Server – използва се за интеграция и оркестрация на бизнес процеси, услуги и системи.
- Microsoft Exchange – позволява координация на съвместната работа в организации. В частност осигурява поддръжката на пощенски услуги (e-mail).
- Microsoft SharePoint Portal Server – позволява сътрудничество и споделяне на информация в реално време. Улеснява конкурентната работа с общи документи и работата в екип.
- Microsoft Host Integration Server – позволява интеграция на стари системи.
- Microsoft Application Center – осигурява хостинг, управление и мониторинг на критични за бизнеса приложения.
- Microsoft Content Management Server – служи за изграждане, поддръжка и управление на уеб съдържание.
- Microsoft Mobile Information Server – позволява интеграция с мобилни приложения.
- Microsoft Internet Security and Acceleration Server – контрол и защита на връзката с Интернет. Предоставя защитна стена (firewall) с възможност за филтриране и анализ на трафика на различни нива.

- Microsoft Commerce Server – използва се за реализация на приложения за електронна търговия.

.NET Framework и Visual Studio .NET 2003

.NET Framework е софтуерна платформа за разработка и изпълнение на .NET приложения. Тя представлява предоставя програмен модел и стандартна библиотека с класове за разработка на приложения и унифицирана среда за изпълнение на управляван код. Поддържа различни езици за програмиране и позволява тяхната съвместна работа.

.NET Framework съществува в два варианта:

- .NET Framework – пълна версия.
- .NET Compact Framework – съкратена версия за изпълнение върху мобилни устройства. Създадена е специално за устройства с ограничени хардуерни ресурси.

Visual Studio .NET 2003 представлява цялостна интегрирана среда за разработка на .NET приложения. Позволява създаване на различни видове приложения, писане на програмен код, изпълнение и дебъгване на приложения, изграждане на потребителски интерфейс и др. VS.NET предоставя единна среда за всички технологии и за всички програмни езици, поддържани стандартно от .NET Framework (C#, VB.NET, C++ и J#).

.NET Building Block Services

.NET Building Block Services са съвкупност от XML уеб услуги, насочени към крайния потребител. Основната им задача е да осигуряват персонализиран достъп до данните на даден потребител по всяко време и от всякакво устройство. За целта се използват отворени стандарти и протоколи за комуникация.

.NET Building Block Services са създадени с цел да позволяват лесна интеграция с други услуги и приложения и да позволяват връзка между тях. Ето няколко области, в които има изградени такива Building Block услуги:

- автентикация – на базата на .NET Passport
- доставка на съобщения
- съхранение на лични потребителски данни – документи, контакти, електронна поща, календар, любими сайтове и други
- съхранение на настройки на приложения, които потребителят използва.

.NET Smart Clients

Smart clients представлява архитектурна концепция, която позволява изграждането на клиентски приложения, които:

- предоставят гъвкав потребителски интерфейс (за разлика от уеб приложенията и WAP приложенията)
- консумират XML уеб услуги (чрез които си осигуряват връзка с останалия свят и обменят данни със сървърите, които съхраняват и обработват техните данни)
- могат да работят в online и offline режим (като синхронизират данните си когато са online)
- имат възможност да се самообновяват (и това може да става автоматично, с минимални усилия от страна на потребителя).

Смарт клиентите предоставят алтернатива на клиент-сървър приложенията и уеб приложенията. Като концепция те не са непременно обвързани с .NET. Има, например, реализация на smart клиент архитектури, базирани на Java платформата.

.NET платформата предоставя специализирана инфраструктура, която подпомага и улеснява реализацията на smart client приложения.

.NET smart клиентите работят както върху обикновени настолни компютри, така и върху различни преносими устройства: мобилни телефони, hand held устройства, вградени системи и т. н.

Основната им задача е да предоставят достъп до информацията, нужна на потребителя, навсякъде, по всяко време и във вид, удобен за потребителя.

.NET Framework и неговия вариант за мобилни приложения .NET Compact Framework предлагат възможности за разработка на smart client приложения за много разнообразни устройства.

Какво е .NET Framework?

До момента направихме преглед на .NET платформата и разгледахме компонентите, от които тя се състои. Сега ще разгледаме в детайли .NET Framework, неговата архитектура и модела за изпълнение на приложения, който тя използва.

.NET Framework е среда за разработка и изпълнение на приложения за .NET платформата. Тя предоставя програмен модел, библиотеки от типове и единна инфраструктура за разработка на приложения и поддържа различни езици за програмиране.

Приложенията, базирани на .NET Framework, се компилират до междинен код (на езика IL) и се изпълняват контролирано от средата за изпълнение на .NET Framework. Компилираният .NET код се нарича още управляван код и може да работи без да се прекомпилира върху различни платформи, за които има имплементация за .NET Framework (Windows, Linux, FreeBSD).

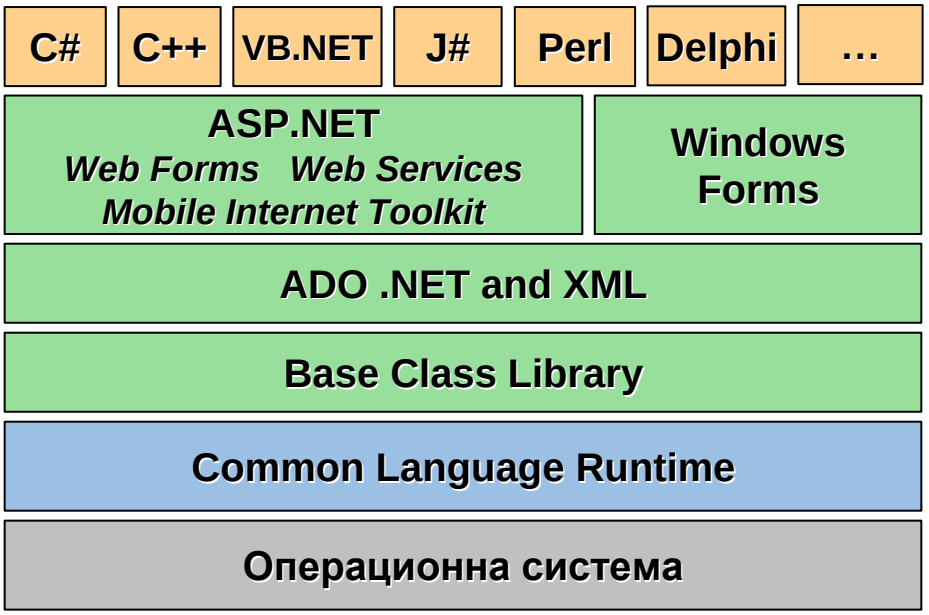
Компоненти на .NET Framework

Можем да разделим .NET Framework на два основни компонента:

- **Common Language Runtime (CLR)** – средата, в която се изпълнява управляваният код на .NET приложенията. Представлява виртуална машина, която контролирано изпълнява .NET кода и осигурява различни услуги, като управление на сигурността, управление на паметта и др.
- **Framework Class Library (FCL)** – представлява основната библиотека от типове, които се използват при изграждането на .NET приложения. Съдържа основната функционалност за разработка, необходима за повечето приложения, като вход/изход, връзка с бази данни, работа с XML, изграждане на уеб приложения, използване на уеб услуги, изграждане на графичен потребителски интерфейс и др. Стандартните класове и типове от FCL можем да използваме навсякъде, където има инсталиран .NET Framework.

Архитектура на .NET Framework

Архитектурата на .NET Framework често пъти се разглежда на нива, както това е направено на следната схема:



Ще разгледаме отделните слоеве един по един и ще обясним тяхната роля в .NET Framework. Ще започнем от най-долния.

Операционна система

Операционната система управлява ресурсите, процесите и потребителите на машината. Тя предоставя и някои услуги на приложенията като например: COM+, MSMQ, IIS, WMI и други.

Средата, която изпълнява .NET приложенията (CLR), е обикновен процес в операционната система и се управлява от нея, както останалите процеси.

Най-често операционната система, която изпълнява CLR е Microsoft Windows, но .NET Framework има имплементации и за други операционни системи (например проектът Mono).

Common Language Runtime

Общата среда за изпълнение Common Language Runtime (CLR) управлява процеса на изпълнение на .NET код. Тя се грижи за заделяне и освобождаване на паметта, управлява конкурентността, грижите за сигурността на приложенията и изпълнява други важни задачи, свързани с изпълнението на кода. Ще обърнем специално внимание на CLR малко по-нататък.

Base Class Library

Base Class Library (BCL) е част от стандартната библиотека на .NET Framework – Framework Class Library (FCL).

BCL представлява богата обектно-ориентирана библиотека с основни класове, които осигуряват базова системна функционалност. BCL осигурява вход-изход, работа с колекции, символни низове, мрежови ресурси, сигурност, отдалечено извикване, многонишковост и др.

Технологиите ADO.NET, XML, ASP.NET и Windows Forms не са част от BCL, тъй като те са по-скоро допълнителни библиотеки, отколкото базова системна функционалност.

ADO.NET и XML

Слоят на ADO.NET и XML предоставя удобен начин за работа с релационни и други бази от данни и средства за обработка на XML. ADO.NET поддържа два модела на работа с данни – свързан и несвързан. XML поддръжката реализира DOM модела и модел, подобен на SAX, за достъп до XML. Ще разгледаме в детайли XML и ADO.NET в темите "Работа с XML" и "Достъп до данни с ADO.NET".

ASP.NET и Windows Forms

ASP.NET и Windows Forms изграждат слоя за интерфейс към крайния потребител на приложенията и ни предоставят богата функционалност за създаване на уеб и Windows базиран потребителски интерфейс, както и уеб услуги. ASP.NET позволява по лесен начин да бъдат изграждани гъвкави динамични уеб сайтове и уеб приложения и уеб услуги. Windows

Forms позволява изграждане на прозоречно-базиран графичен потребителски интерфейс с богати възможности.

ASP.NET и Windows Forms използват компонентно-базирана архитектура и благодарение на нея позволяват изграждане на потребителския интерфейс визуално, чрез сглобяване на компоненти в специално разработени за това редактори, предоставени от средите за разработка. Ще разгледаме в детайли технологиите Windows Forms и ASP.NET в темите "Графичен потребителски интерфейс с Windows Forms", "Изграждане на уеб приложения с ASP.NET" и "Уеб услуги с ASP.NET".

Езици за програмиране

.NET Framework позволява на разработчика да използва различни езици за програмиране, както и да интегрира в едно приложение компоненти, разработвани на различни езици. Възможно е дори клас, написан на един език, да бъде наследен и разширен от клас, написан на друг език.

Microsoft .NET Framework поддържа стандартно езиците C#, VB.NET, Managed C++ и J#, но трети доставчици предлагат допълнително .NET версия на още много други езици, като Pascal, Perl, Python, Fortran, Cobol и други.

Съвместимостта на езиците за програмиране в .NET Framework се дължи на архитектурни решения, които ще разгледаме в детайли след малко.

Common Language Runtime

След като се запознахме накратко с архитектурата на .NET Framework, нека сега разгледаме в детайли и най-важният компонент от нея – CLR.

Common Language Runtime (CLR) е сърцето на .NET Framework. Той представлява **среда за контролирано изпълнение на управляван код**. На практика CLR е тази част от .NET Framework, която изпълнява компилираните .NET програми в специална изолирана среда.

В своята същност CLR представлява виртуална машина, която изпълнява инструкции, на езика IL (Intermediate Language), езикът до който се компилират всички .NET езици. CLR е нещо като виртуален компютър, който обаче не изпълнява асемблерен код за процесор Pentium, AMD или някакъв друг, а IL код.

Има голямо сходство между .NET CLR и Java Virtual Machine, но между двете технологии и много разлики. По предназначение те служат за едно също нещо – да изпълняват код за някакъв виртуален процесор. В .NET това е IL кода, а при Java платформата – т. нар. Java bytecode. Основната разлика между IL и bytecode е, че IL е език от по-високо ниво, а това позволява да бъде компилиран много по-ефективно от Java bytecode.

Задачи и отговорности на CLR

Отговорностите на CLR включват:

- **Изпълнение на IL кода**. Реално IL инструкциите, преди да бъдат изпълнени за първи път, се компилират до инструкции за текущия процесор и след това се изпълняват от системния процесор. Този процес на междинно компилиране до машиннозависим (native) код се нарича **JIT компилация** (Just-In-Time compilation).
- **Управление на паметта и ресурсите** на приложенията. CLR включва в себе си система за заделяне на памет и система за почистване на неизползваната памет и ресурси (т. нар. **garbage collector**). Управлението на паметта при .NET приложенията се извършва в голяма степен автоматизирано и в повечето случаи програмистът не трябва да се грижи за освобождаване на заделената памет. Ще разгледаме в детайли как .NET Framework управлява паметта в темата "Управление на паметта и ресурсите".
- **Осигуряване безопасността на типовете**. .NET Framework е среда за контролирано изпълнение на програмен код (**managed execution environment**). Тя не позволява директен достъп до паметта, не позволява директна работа с указатели, не позволява преобразуване от един тип към друг, който не е съвместим с него, не позволява излизане от границите на масив, както и всякакви други опасни операции. По тази причина .NET се нарича управлявана среда – защото тя управлява изпълнението на кода и по този начин предпазва програмите от много досадни проблеми, които възникват при неуправляваните среди.
- **Управление на сигурността**. NET Framework има добре изградена концепция за сигурност на различни нива. От една страна .NET приложенията могат да се изпълняват с различни права. Правата могат да се задават от администраторите чрез т. нар. политики за сигурност. CLR следи дали кодът, който се изпълнява, спазва зададената политика за сигурност и не позволява тя да бъде нарушена. Тази техника се нарича "code access security". От друга страна .NET Framework поддържа и средства за управление на сигурността, базирана на роли (role-based security). Ще разгледаме в детайли всички тези техники и средства в темата "Сигурност в .NET Framework".
- **Управление на изключенията**. .NET Framework е изцяло обектно-ориентирана среда за разработка и изпълнение на програмен код. В нея механизмът на изключенията е залегнал като основно средство за управление на грешки и непредвидени ситуации. Една от задачите на CLR е да се грижи за изключенията, които възникват по време на изпълнение на кода. При настъпване на изключение CLR има грижата да намери съответния обработчик и да му предостави управлението. Ще разгледаме в детайли всичко това в темата "Управление на изключенията в .NET".
- **Управление на конкурентността**. CLR контролира паралелното изпълнението на нишки (threads) като за целта си взаимодейства с

операционната система. Повече за работата с нишки ще научим в темата "Многонишково програмиране и синхронизация".

- **Взаимодействие с неуправляван код**. CLR осигурява връзка между управляван (.NET) код и неуправляван (Win32) код. За целта той изпълнява доста сложни задачи, свързани с конвертиране на данни, синхронизация, прехвърляне на извиквания, взаимодействие с компонентния модел на Windows (COM) и много други. Ще разгледаме в детайли тези проблеми в темата "Взаимодействие с неуправляван код".
- **Подпомагане процесите на дебъгване (debugging) и оптимизиране (profiling)** на управлявания код. CLR осигурява инфраструктура и средства за реализацията на дебъгване и оптимизиране на кода от външни специализирани програми.

Управляван код

Управляваният код (managed code) е кодът, който се изпълнява от CLR. Той представлява поредица от IL инструкции, които се получават при компилацията на .NET езиците. По време на изпълнение управляваният код се компилира допълнително до машиннозависим код за текущата платформа и след това се изпълнява директно от процесора.

Управляван код и неуправляван код

Управляваният код (.NET код) се различава значително от неуправляваният код (например Win32 кода).

Управляваният код е машиннонезависим, т. е. може да работи на различни хардуерни архитектури, процесори и операционни системи, стига за тях да има имплементация на CLR.

Неуправляваният код е машиннозависим, компилиран за определена хардуерна архитектура и определен процесор. Например програмите, написани на езика C, се компилират до неуправляван код за определена архитектура.

Ако компилираме една C програма за Embedded Linux върху платформа StrongARM, ще получим неуправляван машиннозависим (native) код за Linux за тази платформа. Кодът ще съдържа инструкции за микропроцесор StrongARM и ще използва системни извиквания към операционната система Embedded Linux. Съответно на друга платформа няма да може да работи без прекомпилация на сорс кода на C програмата.

По същия начин, ако компилираме една C програма за Windows върху архитектура x86, ще получим неуправляван код за процесор x86 (примерно Pentium, Athlon и т.н.), който използва системни извиквания към Windows. Този код се нарича Win32 код и може да работи само върху 32-битова Windows операционна система. За да се стартира върху друга платформа, трябва да се компилира.

При управлявания код нещата стоят по различен начин. Ако компилираме една C# програма за платформа .NET Framework 1.1, ще получим управляван, машиннонезависим IL код, който може да работи върху различен хардуер. Кодът реално ще е компилиран за платформа CLR 1.1 и ще се състои от IL инструкции за виртуалния процесор на CLR и ще използва системни извиквания към .NET Base Class Library.

Управляваният код лесно може да бъде пренесен върху различни платформи без да се променя или прекомпилира. Така например програма на C#, която е компилирана под Windows до управляван IL код, може да се изпълнява без промени както върху Windows под .NET Framework, така и върху Linux под Mono, а също и върху мобилни устройства под Windows CE и .NET Compact Framework.

Метаданните в управлявания код

Управляваният код се самоописва чрез **метаданни** и носи в себе си описание на типове данни, класове, интерфейси, свойства, полета, методи, параметри на методите и други, както и описание на библиотеки с типове, описание на изисквания към сигурността при изпълнение и т. н. Това дава голяма гъвкавост на разработчика и възможност за динамично зареждане, изследване и изпълнение на функционалност, компилирана като управляван (IL) код.

Неуправляваният код стандартно не съдържа метаданни и това силно затруднява динамичното зареждане и изпълнение на неуправлявана функционалност.

Управляваният код е обектно-ориентиран

Управляваният код задължително е обектно-ориентиран, докато за неуправлявания няма такова изискване. Всички .NET езици са обектно-ориентирани. Всички .NET програми се компилират до класове и други типове от общата система от типове на .NET Framework. Всички данни, използвани от управлявания код, са наследници (в смисъла на обектно-ориентираното програмиране) на базовия тип **`System.Object`**. Ще разгледаме това в подробности в темата "Обща система от типове".

Управляваният код е високо надежден

Управляваният код е защитен от неправилна работа с паметта и типовете и това го прави по-сигурен и високо надежден. Управляваният код не може да извършва неправилен достъп до паметта, достъп до чужда памет и неправилна работа с типове. Това предпазва програмисти от много досадни проблеми, присъщи при писането на неуправляван код, като загуба на памет, достъп до неинициализирана памет, повторно освобождаване на памет, работа с невалиден указател и т.н.

Управляваният код интегрира различни езици

До управляван код се компилират всички .NET езици. Това дава възможност за широко взаимодействие между код, писан на различни езици за

програмиране. Възможно е дори клас, написан на един .NET език, да бъде наследен и разширен от клас, написан на друг .NET език.

За .NET Framework няма значение на какъв език е бил написан кода преди да бъде компилиран. Всичкият код се компилира до IL и се изпълнява от CLR по еднакъв начин.

Управление на паметта

Управлението на паметта е една от важните задачи на CLR. Идеята за автоматизирано управление на паметта е залегнала в .NET Framework на дълбоко архитектурно ниво. Целта е да се улесни разработчика като се освободи от досадната задача сам да следи за освобождаването на заделената памет.

CLR, като средата за изпълнение, управлява заделянето на памет, инициализирането й, и автоматичното й освобождаването посредством garbage collector.

Динамично заделените обекти се разполагат в динамичната памет, в тъй наречения "managed heap". След като техния живот завърши и те вече не са необходими на приложението, системата за почистване на паметта (garbage collector) освобождава заеманата от тях памет автоматично. По този начин се избягват най-често срещаните проблеми като загуба на памет и достъп до освободена или неинициализирана памет. Повече за управлението на паметта в .NET Framework ще научим в темата "Управление на паметта и ресурсите".

Важна особеност при работата с управляван код е, че при него няма указатели. Вместо указатели се работи с референции, които са силно типизирани и се управляват автоматично. **Референцията (reference)** прилича на указател, но не е просто адрес в паметта, а има тип, т. е. тя е указател към определен тип данни и не може да сочи към място в паметта, където няма инстанция на този тип.

Intermediate Language (IL)

Междинният език Intermediate Language (IL), е език за програмиране от ниско ниво, подобен на асемблерните езици. За разлика от тях, обаче, IL е от много по-високо ниво, отколкото асемблерите за съвременните микропроцесори.

IL е обектно-ориентиран език. Той разполага с инструкции за заделяне на памет, за създаване на обект, за предизвикване и обработка на изключения, за извикване на виртуални методи и други инструкции, свързани с обектно-ориентираното програмиране.

Тъй като не е процесорно-специфичен, IL предоставя голяма гъвкавост и възможност за изпълнение на кода върху различни платформи чрез компилиране до съответния за платформата машинен език.

Възможна е и предварителна компилация до код за текущата платформа, но тази техника не носи голяма полза и рядко се използва.

Имплементацията на IL в .NET Framework се нарича **MSIL (Microsoft Intermediate Language)**. IL може да има и други имплементации в други платформи и среди за изпълнение на .NET код.

Езикът IL е стандартизиран от организацията ECMA и в съответния стандарт се нарича **CIL (Common Intermediate Language)**.

Често пъти термините IL и MSIL се използват като взаимозаменяеми и затова винаги трябва да имате предвид, че става въпрос за кода, който се изпълнява от CLR – машинният код, получен при компилацията на .NET езиците.

Intermediate Language (IL) – пример

За да илюстрираме по-добре казаното до тук, нека разгледаме една проста програмка, написана на MSIL – класическият пример "Hello world!":

```
.method private hidebysig static void Main() cil managed
{
  .entrypoint
  // Code size       11 (0xb)
  .maxstack  8
  ldstr      "Hello, world!"
  call       void  [mscorlib]System.Console::WriteLine(string)
  ret
} // end of method HelloWorld::Main
```

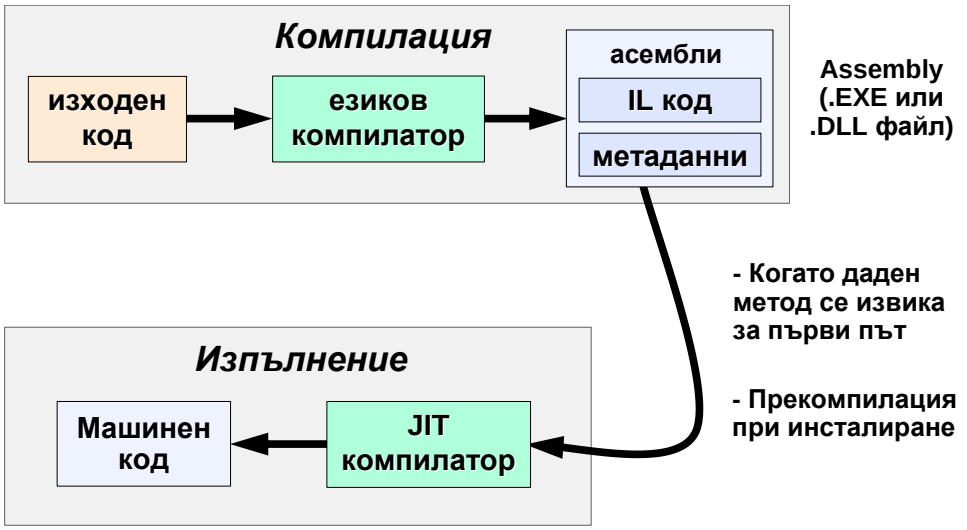
Всичко, което прави тази MSIL програма, е да изведе съобщението "Hello, world!" на конзолата. Тя дефинира един статичен метод без параметри с име **`Main`**, в който извиква с параметър "Hello, world!" статичния метод **`WriteLine()`** от класа **`System.Console`**, който отпечатва посочения текст.

Компилация и изпълнение

Вече споменахме няколко пъти междинния код IL и обяснихме, че .NET езиците (C#, VB.NET и т. н.) се компилират до него, а след това полученият код се изпълнява от CLR.

Сега ще разгледаме детайлно процеса на компилиране и изпълнение на .NET приложенията. Ще изясним как се извършва компилирането на програми от високо ниво, как се получава IL код, как този код се записва в специален файлов формат (асемблы) и как след това компилираните асемблита се изпълняват от CLR като се компилират междувременно до машинен код от JIT компилатора.

Целият този процес е изобразен схематично на фигурата:



Изходният код на .NET програмите може да е написан на предпочитания от нас .NET език, например C#, VB.NET, Managed C++ или друг. За да го компилираме до IL управляван код, използваме компилатора за съответния език. В резултат получаваме асемблы.

Асемблито представлява изпълним файл, съдържащ .NET управляван код и метаданни, които описват съдържанието на асемблито. Метаданните съдържат имената на класовете и типовете в асемблито, информация за членовете на класовете (методи, полета, свойства и други).

Едно асемблы може да бъде изпълним файл (`.exe` файл) или динамична библиотека (`.dll` файл). Изпълнимите файлове съдържат допълнителна информация, която подпомага началното им стартиране (например входна точка на изпълнение).

При изпълнение на дадено асемблы CLR го зарежда в паметта и анализира метаданните му. Извършват се различни проверки на кода – дали е коректен спрямо IL стандарта, дали има необходимите права за изпълнение и др.

След това управляваният IL код преминава през специфичния за текущата платформа JIT компилатор и се компилира до машинен код за текущия процесор. Компилираният вече код след това се изпълнява директно от процесора.

JIT компилаторът не компилира в началото цялото асемблы, а само методът, от който започва изпълнението му. След това при опит за изпълнение на некомпилиран метод, той се компилира. Така кодът се компилира само при нужда и това осигурява добро бързодействие. Забавянето е незначително и скоростта на изпълнение на управлявания код на практика е почти еднаква със скоростта на изпълнение на неуправлявания код.

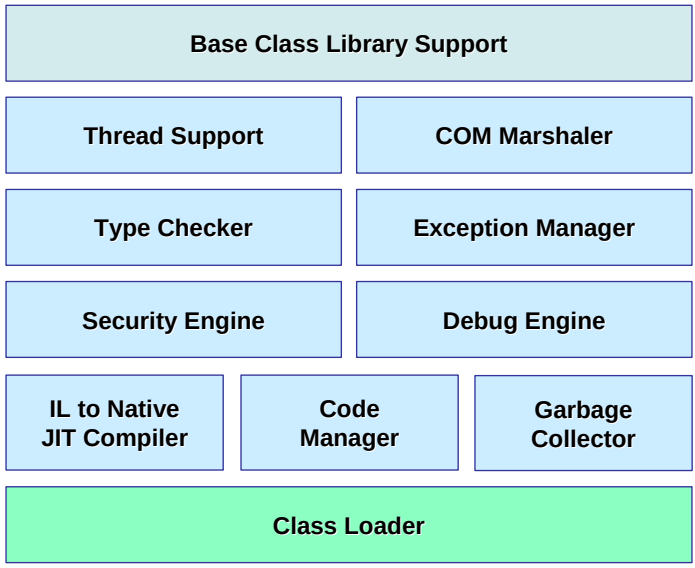
Предимството на JIT компилацията е, че може да оптимизира кода за текущата хардуерна платформа по най-добрия начин. Например ако е наличен най-мощният процесор на Intel или AMD и CLR поддържа този процесор, той ще компилира IL кода по начин, оптимизиран специално за него, и ще използва пълните му възможности. При неуправляваният код това не е възможно, защото кодът се компилира така, че да работи върху всички процесори, без да използва пълните възможности на текущата хардуерна платформа. По тази причина в някои случаи управляваният код може да е дори по-бърз от неуправлявания въпреки нуждата от JIT компилация, която отнема време.

Когато разполагаме с компилирано асембли и искаме да го изпълним, имаме право на избор кога да компилираме IL кода до машинен код. Това може да стане по време на изпълнение (посредством JIT компилатора) и предварително (с прекомпилация за текущата платформа).

Прекомпилацията на асемблита се извършва с инструмента **`ngen.exe`**, който е стандартна част от .NET Framework.

Архитектура на CLR

Общата среда за изпълнение CLR се състои от доста модули, всеки от които изпълнява конкретна задача. Схематично архитектурата можем да представим по следния начин:



Ще разгледаме всеки от посочените компоненти съвсем накратко, тъй като функциите им са от много ниско ниво и рядко ще ни се налага да взаимодействаме директно с тях:

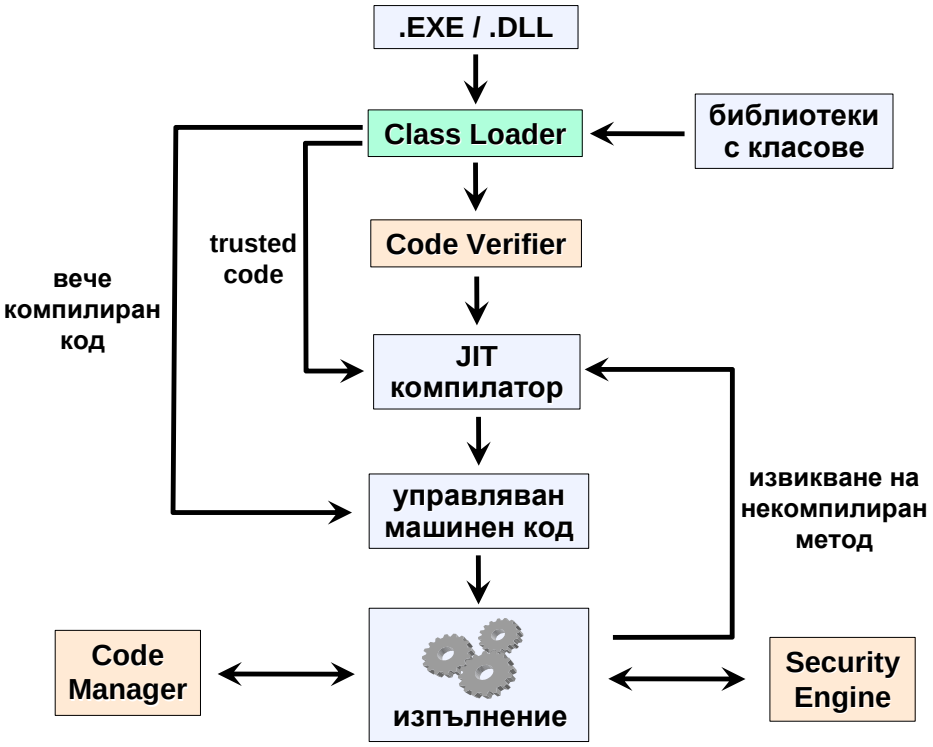
- Base Class Library Support – предоставя системни услуги, необходими за работата на Base Class Library (BCL).

- Thread Support – предоставя услуги за манипулация на нишки в .NET приложенията – създаване на нишка, управление на състоянието на нишка, синхронизация и др.
- COM Marshaler – грижи се за комуникацията с COM обекти. Осигурява извикването на COM сървъри от .NET код и извикването на .NET код от COM. Негова грижа са прехвърлянето на заявки, преобразуването на данни, управлението на жизнения цикъл на COM обектите и др.
- Type Checker – осъществява проверка на типовете за съответствие при извикване и поддържа класовите йерархии.
- Exception Manager – грижи се за управление на изключенията – предизвикване на изключение, прихващане, обработване и др.
- Security Engine – отговаря за проверките на сигурността при изпълнение на кода.
- Debug Engine – осигурява функционалност, свързана с дебъгването и оптимизирането на управляван код.
- JIT Compiler – един от най-важните модули – по време на изпълнение компилира IL кода в специфичен за процесора код.
- Code Manager – управлява изпълнението на кода.
- Garbage Collector – управлява паметта автоматичното почистване на паметта и ресурсите. Контролира живота на обектите.
- Class Loader – служи за зареждане на класове и типове. Използва се при началното изпълнение на приложението, както и при динамично зареждане на код по време на изпълнение.

Как CLR изпълнява IL кода?

Нека сега разгледаме по-подробно как CLR изпълнява IL кода. Изпълнението на кода, както можем да видим от схемата по-долу, е итеративен процес, който се състои от много стъпки.

При изпълнение на метод от едно асембли Class Loader подсистемата на CLR зарежда всички нужни за неговата работа класове и типове. В зависимост от това дали кодът е вече компилиран до машинен или не Class Loader предава кода за директно изпълнение или го компилира с JIT компилатора (при първо извикване на всеки метод).



Преди JIT компилацията се извършва процес, известен като верификация. Той проверява дали IL кодът е безопасен – дали не се опитва да осъществява директен достъп до паметта, дали не се опитва да заобикаля механизмите за сигурност и т. н. Ако системният администратор е определил кода за сигурен (trusted) неговото верифициране може да бъде се прескочено.

JIT компилаторът създава специфичен за машината код (native код), който се изпълнява директно от процесора. Този машинен код съдържа в себе си много допълнителни инструкции, чрез които си взаимодейства със CLR. Целта е кодът да се изпълнява по контролиран начин, за да не нарушава принципите за сигурност и надеждност, но без да се забавя излишно заради всички допълнителни проверки.

При изпълнението на кода, при достъп до ресурси, при извикване на системни библиотеки и в много други случаи се извършват проверки на сигурността (чрез т. нар. security engine).

Ако трябва да бъде извикан некомпилиран метод, този метод се връща в JIT компилатора и така се затваря цикълът на компилация до машинен код. Като резултат от описания алгоритъм не се налага компилиране на един и същ метод повече от веднъж и освен това, ако някой метод не се извиква никъде в приложението, той въобще не се компилира от JIT компилатора.

Асемблита

Асемблитата са най-малката самостоятелна градивна единица в .NET Framework. Те представляват наследници на познатите ни `.exe` и `.dll` файлове и съдържат **IL изпълним код**, **метаданни** и **ресурси**:

За разлика от неуправляваните изпълними файлове, асемблитата са самоописващи се и носят в себе си информация за всички класове, типове и ресурси, които съдържат, както и информация за сигурността, за зависимост от външни компоненти и др. Тази информация се нарича метаданни.

Асемблитата имат собствена версия и дефинират изисквания към правата, свързани със сигурността, на потребителя или процеса, който ги изпълнява. Те могат да имат и цифров подпис, положен от създателя им, чрез който се осигурява повишена сигурност.

Проблемът "DLL Hell"

С вграждането на версия в самия файл на асемблито се разрешава проблемът, известен като "DLL Hell". Той се състои в следното: Когато няколко приложения зависят от един и същ споделен файл и от точно определена негова версия, с досегашните технологии в Windows при поява на нова версия на този файл, старият файл се презаписва и на практика няма гаранция, че новата версия е 100% съвместима със старата. В резултат по "магически" начин някое от приложенията, което е използвало старата версия, престава да работи.

Например при Win32 приложенията може да се случи при инсталиране на нов софтуер част от старите приложения, които са работели до този момент, да спрат да работят. Причината е в това, че новият софтуер презаписва някоя от библиотеките, които старите приложения са използвали, с по-нова версия, която може да не е съвместима със старата.

Тъй като при асемблитата версията се задава за всяко едно от тях и се записва освен в метаданните на асемблито и в неговото файлово име, при поява на нова версия не се появяват конфликти между приложенията.

Всяко асембли може да посочи точно кое асембли му е необходимо и точно в коя негова версия. Освен, че могат да съществуват едновременно няколко различни версии на едно асембли, те могат и да се изпълняват едновременно. Така е възможно в един и същи момент да работят и

старите и новите приложения и всяко приложение да използва версията на общите асемблита, с която е предвидено да работи.

Всяко асембли съдържа т. нар. манифест, в който се описват зависимостите, които има с други асемблита. В него се определя и политиката за избор на версия, в случай че има повече от една за някое от реферираните асемблита.

Метаданни

Както вече споменахме, всички асемблита съдържат **метаданни**. Метаданните описват различни характеристики на асемблитата и съдържимото в тях:

- име на асемблито (например **`System.Windows.Forms`**)
- версия, състояща се от 4 числа (например **`1.0.5000.0`**)
- локализация, описваща език и култура (например неутрална или en-US или bg-BG)
- цифров подпис на създателя (незадължителен)
- изисквания към правата за изпълнение
- зависимости от други асемблита (описани с точното име и версия)
- експортирани типове
- списък със дефинираните класове, интерфейси, типове, базови класове, имплементирани интерфейси и т.н.
- списък с дефинираните ресурси

Освен тези данни за всеки клас, интерфейс или друг тип, който е дефиниран в асемблито, се съдържа и следната информация:

- описание на член-променливите, свойствата и методите в типовете
- описание на параметри на методите, връщана стойност на метода за всеки метод
- приложени атрибути към асемблито, методите и другите елементи от кода

IL код

Във всяко асембли може да има секция, в която се намира неговият изпълним код (IL кодът). Тази секция не е задължителна. Вече разгледахме какво представлява IL кодът и как се изпълнява, така че няма да се спираме отново на това.

Ресурси

В ресурсната секция на асемблито могат да бъдат добавяни различни ресурси (иконки, картинки, локализирани низове и др.), необходими на

приложението. Ресурсите могат да се пакетират във файла на асемблито, заедно с изпълнимия код и могат да се извличат от него по време на изпълнение. Тази секция не е задължителна.

За удобство имаме възможност да създаваме и асемблита, които се състоят от няколко файла, както и сателитни асемблита с различна култура. Ще разгледаме асемблитата по-детайлно в темата "Асемблита и разпространение".

Разгръщане на асемблита

Тъй като асемблитата са основната единица за разгръщане (deployment) в .NET Framework, ще се спрем накратко върху различните видове асемблита според начина им на разгръщане – **частни** и **споделени**.

Частните асемблита (private assemblies) се използват само от едно приложение и се записват в неговата директория или в нейна поддиректория. Те са лесни за разгръщане тъй като могат да се разпространяват чрез просто копиране и вмъкване (copy/paste). При тях контролът на версиите е по-лесен и не се изисква цифров подпис на създателя или силно име.

Споделените асемблита от своя страна са достъпни за всички приложения. Те се инсталират в специална област, наречена Global Assembly Cache (GAC). Всяко приложение, което реферира външно асембли търси споделените асемблита в тази област. Това поведение може да се контролира чрез манифеста, който задава правилата за търсене на нужните асемблита и версии. За да се определи уникално всяко асембли използва т. нар. **силно име (strong name)**. То включва:

- име на асемблито
- версия
- локализация
- цифров подпис на създателя

Пример за силно име на асембли е идентификаторът:

```
["myDll, Version=1.0.0.1, Culture=neutral, PublicKeyToken=
9b35aa32c18d4fb1"]
```

Повече за разгръщане на приложения и асемблита можете да намерите в темата "Асемблита и разпространение".

.NET приложения

.NET приложенията се състоят от едно или повече асемблита, в които се съдържат техният код и ресурси. Те представляват изпълними единици, които могат да бъдат конфигурирани.

В зависимост от вида си .NET приложенията могат да бъдат самостоятелни или обвързани с други услуги или приложения. Например уеб приложенията не са самостоятелни и се изпълняват в средата на ASP.NET, докато конзолните приложения могат да се изпълняват самостоятелно.

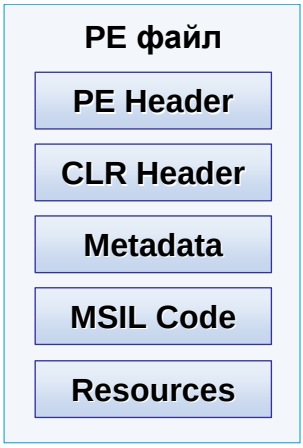
За разлика от повечето Win32 приложения, .NET приложенията могат да бъдат инсталирани с просто копиране (XCOPY deployment), без да се налага регистриране на отделните им компоненти (регистрирането се налага само, ако искаме да позволим неуправлявани компоненти да могат да достъпват нашите асемблита). При .NET приложенията не се използва Windows Registry за регистрация на компонентите.

Всяко приложение използва собствена политика за зареждане на свързаните с него асемблита и намирането на нужната им версия. При липса на изрично указана такава първо се търси подходящо асембли в директориите на приложението и после в GAC. Всяко едно приложение може да използва различна версия на дадено асембли без да се влияе от останалите и, както вече споменахме, новите версии не предизвикват конфликти.

Поради всички посочени качества инсталирането, поддържането, обновяването и деинсталирането на .NET приложения е лесно и безопасно за останалите приложения.

Преносими изпълними файлове

В Windows и в .NET Framework се въвежда понятието "преносим изпълним файл" (Portable Executable или PE). Това са файлове, които съдържат информация за себе си, съдържат своя изпълним код и необходимите за работата си ресурси. Структурата на PE файловете може да се онагледи със следната фигура:



PE хедърът съдържа описание за вида на самия PE файл – дали той е изпълним или е библиотека с типове.

След него CLR хедърът дава нужната на CLR информация за изпълнение на самото асембли.

Останалите елементи са ни вече познати от структурата на асемблитата и затова няма да се спираме на тях отново.

Понятието PE представлява обобщение на двата файлови формата – **`.exe`** и **`.dll`** (става дума единствено за Windows платформи – при другите имплементации на CLI е много вероятно тези формати да се описват по друг начин.)

Application domains

Application domain (домейн на приложението) е ново понятие, което се въвежда с .NET Framework. То представлява допълнително ниво на изолация между отделни .NET приложения, изпълнявани в един и същ процес на операционната система.

За да се ограничат възможните проблеми, свързани с манипулиране на паметта, в операционната система всеки процес разполага със собствена памет, с която работи, и няма право да чете или пише в паметта на друг процес. Ако се налага такова взаимодействие, то се извършва индиректно например чрез прокси обекти.

Както вече знаем, всяко .NET приложение изисква CLR да бъде зареден в паметта. Ако трябва да го зареждаме за всяко .NET приложение, което стартираме, това ще предизвика голямо ненужно натоварване и неефективно използване на ресурсите.

Като решение на този проблем идва концепцията за обединяване на няколко .NET приложения в един процес от операционната система. Това, обаче крие рискове приложенията да си пречат едно на друго. Необходим е начин за изолиране на приложенията едно от друго в рамките на процеса. Точно такава е задачата на домейните на приложенията (application domains) – те ни позволяват да изпълняваме няколко приложения в един и същ процес и същевременно ни дават пълна изолация между тях. По този начин намаляваме броя на процесите, спестяваме разход на процесорно време за зареждане на CLR и прехвърляне между процесите и намаляваме количеството на използваната памет и елиминираме повторното зареждане на едни и същи библиотеки.

.NET Framework използва вътрешно домейни на приложението за много цели, например за да изолира едно от друго отделните ASP.NET уеб приложения в рамките на един уеб сървър.

Интеграция на езиците за програмиране

Една от най-добрите черти на .NET Framework е възможността за интеграция на множество езици за програмиране. Тя позволява да работим на предпочитания от нас език и да не губим възможността за използване на други езици в рамките на нашето решение.

За .NET Framework няма значение на какъв език е написан даден клас или компонент, стига езикът да поддържа общата езикова спецификация – CLS (Common Language Specification), т.е. да е един от .NET езиците.

Интеграцията на различни езици в .NET Framework е възможна благодарение на три важни стандарта:

- Common Language Specification (CLS)
- Intermediate Language (IL)
- Common Type System (CTS)

Ще разгледаме тези стандарти един по един с изключение на IL, тъй като вече се запознахме с него.

Common Language Specification (CLS)

CLS дефинира общите и задължителни характеристики, които един програмен език трябва да притежава, за да бъде съвместим с останалите .NET езици. Тази спецификация има за цел да минимизира разликите между .NET езиците.

CLS, например, налага ограничението да се прави разлика между малки и главни букви в имената на типовете и техните публични членове, методи, свойства и събития. Ако нарушим това правило, нашият код ще се компилира, но ще загуби съвместимостта си с CLS и другите .NET езици.

Друго ограничение, което CLS налага, е езиците да бъдат обектно-ориентирани. Това означава, че за да бъде направен даден език съвместим с CLS и .NET Framework, той трябва да бъде разширен да поддържа класове, интерфейси, свойства, изключения и всички останали елементи на обектно-ориентираното програмиране с .NET.

Повечето .NET езици поддържат много повече възможности от тези, които изисква CLS. Поради това трябва да сме внимателни при създаването на класове и други типове и да подхождаме с ясната идея дали искаме те да са CLS съвместими или не.

Common Type System (CTS)

Общата система от типове в .NET Framework представлява формална спецификация на типовете данни, използвани в различните .NET езици за програмиране. CTS описва различните .NET типове (примитивни типове данни, класове, структури, интерфейси, делегати, атрибути и др.). В CTS се описват съдържанието и начина на дефиниране на типовете, модификаторите за достъп, начините за наследяване, времето на живот на обектите и много други технически характеристики.

CTS ни гарантира съвместимостта на данните между отделните езици за програмиране. Например типът **`String`** в C# е същия като **`String`** във Visual Basic .NET. Това позволява кодът, писан на различни езици, да си обменя свободно данни, защото данните са съвместими с CTS.

CTS дефинира двата типа обекти – **референтни** и **стойностни**, според това как се пазят в паметта и как се манипулират. CTS налага задължението всички типове да наследяват системния тип `System.Object`, дори и примитивните. Благодарение на това извикването "`5.ToString()`" е напълно валидно в езика C#.

Ще опишем съвсем накратко референтните и стойностните типове в CTS, а в по-големи детайли относно тях ще навлезем в темата "Обща система от типове".

Референтни типове

Референтни типове (**reference types**) са всички класове, масиви, интерфейси и делегати. Класът `String` също е референтен тип. Техните инстанции представляват типово-обезопасени указатели към паметта, в която са записани данните за определен обект.

Инстанциите на референтните типове се съхраняват в динамичната памет (**managed heap**) и подлежат на почистване от системата за събиране на боклука (**garbage collector**). При предаване като параметър, те се предават по референция (адрес).

Стойностни типове

Стойностни типове (**value types**) са структурите и примитивните типове (като `int`, `float`, `char` и други). Този тип обекти се съхраняват в стека и се унищожават при излизане от обхват. При предаване като параметър, се предават по стойност (освен, ако изрично не е указано друго).

Common Language Infrastructure (CLI)

Спецификацията за общата инфраструктура на .NET езиците CLI (Common Language Infrastructure) е стандартизираната част от CLR. Нейната цел е още по-мащабна от идеята да се интегрират различните езици за програмиране – става дума за междуплатформена съвместимост. За целта тя е стандартизирана от организациите ECMA и ISO (стандарт ISO 23271:2003).

В CLI се описва как приложения написани на различни езици да могат да се изпълняват в различни среди без да се налага да се променят или прекомпилират.

CLI стандартизира следните компоненти на .NET Framework:

- Common Language Specification (CLS)
- Common Type System (CTS)
- Common Intermediate Language (CIL)
- Начина за управление на изключения в .NET
- Форматите за асемблита и метаданни

- Части от .NET Framework Class Library

Имплементацията на CLI стандарта за Windows е Microsoft .NET Framework, а за UNIX и Linux е Mono. С учебна цел Майкрософт разпространяват официално имплементация на CLI с отворен код, т. нар. Shared Source CLI (http://msdn.microsoft.com/net/sscli/).

.NET езиците

Microsoft предлагат компилатори и поддръжка във Visual Studio .NET 2003 за следните езици:

- **C#** - препоръчителният език за програмиране под .NET Framework. Съвременен обектно-ориентиран език, подобен на C++ и Java, разработен специално за .NET Framework.
- **Visual Basic .NET** – обновена версия на езика Microsoft Visual Basic, адаптирана към .NET Framework.
- **C++** (managed/unmanaged) – езикът C++ по идея е език от доста по-ниско ниво в сравнение със C# и VB.NET. Той е адаптиран към .NET Framework чрез множество разширения, допълнения и ограничения и е наречен Managed C++. Езикът продължава да съществува и като неуправляван език, който не е съвместим с .NET и се нарича Unmanaged C++.
- **J#** – езикът J# е създаден за да позволи по-лесното прехвърляне на Java приложения към C#. Той спазва синтаксиса на езика Java, на използва както стандартните библиотеки на Java платформата, така и стандартните библиотеки на .NET (Framework Class Library).
- **JScript.NET** – езикът JScript.NET е представител на слабо типизираните скриптови езици от фамилията ECMAScript (като JavaScript, VBScript и JScript), но е адаптиран към .NET Framework. Използва се за изпълнение на скриптове в някои уеб браузъри и някои други приложения.

Допълнително освен стандартните .NET езици трети доставчици са разработили съвместими с .NET Framework компилатори за Perl (ActiveState Perl for .NET), Python, Pascal (Borland Delphi 2005), Smalltalk, APL, COBOL, Eiffel, Haskell, Scheme и др.

Можем да използваме най-удобния ни език и да го смесваме с други езици в рамките на едно приложение. Имаме възможност да наследяваме безпроблемно типове, дефинирани на друг програмен език. Дори можем да използваме ефективно системата за изключения и тяхната обработка между езиците.

Интеграцията на езиците за програмиране в .NET Framework е вградена и не се налага да правим "акробатики" за да я използваме. Това е възможно поради единните система от типове, програмен модел и библиотеки от класове.

Езикът C#

Както вече споменахме, C# е препоръчваният език за програмиране за .NET Framework. Този език е специално проектиран от архитектите на .NET Framework и е съобразен с особеностите на платформата още по време на дизайна. Именно по тази причина в настоящата книга всички примери и програмен код са написани на C#.

C# компилаторът е част от стандартния пакет на Microsoft .NET Framework SDK. C# е нов език, който се появява за пръв път в .NET и представлява смесица между C++ и Java, с елементи от Delphi. Проектиран е от екипа на Андерс Хейлсбърг, създателят на средата за бърза разработка на приложения Delphi, който е работил дълги години като архитект в Borland, а по-късно се присъединява към Microsoft.

C# е съвременен обектно-ориентиран език, силно типизиран, с широка поддръжка на идеите на компонентно-ориентирания подход за разработка. C# поддържа синтаксис за дефиниране и използване на свойства и събития, които играят важна роля при дефинирането и използването на компоненти.

C# е наследник на езика C++, но не наследява от него всичко, а само част от синтаксиса и някои негови силни страни (например предефинирането на оператори). По идея C# е проектиран да бъде лесен за използване като Java, но мощен като C++ и до голяма степен тази идея е осъществена.

В C# е премахната нуждата от допълнителни файлове като хедъри, IDL дефиниции и други, познати ни в повечето езици като C и C++. Езикът няма никакви ограничения в употребата си – еднакво добре можем да програмираме Windows, уеб или конзолни приложения, услуги (services) или библиотеки.

Заради силната типизация в C# всичко е обект – всеки един от типовете, дефинирани било в .NET Framework, било от нас, директно или индиректно наследяват базовия тип **`System.Object`**.

Самият език C# е стандартизиран от ECMA и ISO още преди да бъде реализирана финалната му версия в .NET Framework.

Езикът C# – пример

Както вече обяснихме, настоящата книга разглежда работата с .NET Framework в контекста на езика C#, така че от тук нататък често ще срещаме правоъгълни области с примерен код, като следващата:

```
using System;

namespace HelloCSharp
{
   class HelloCSharp
   {
```

```
      static void Main()
      {
         Console.WriteLine("Hello, C#!");
      }
   }
}
```

Примерът дефинира нашата първа програма на C# – класическата програмка "Hello, World!", която е адаптирана за C# и се е превърнала в "Hello, C#!". Сега няма да обясняваме в детайли как работи тя, защото ще направим това по-късно, в темата "Въведение в C#".

Framework Class Library

Framework Class Library (FCL) е стандартната библиотека на .NET Framework. В нея се съдържат няколко хиляди дефиниции на типове, които предоставят богата функционалност.

FCL съдържа средства, които позволяват на програмистите да разработват различни видове приложения:

- Windows приложения с прозоречно-базиран графичен потребителски интерфейс
- Уеб-базирани приложения
- Конзолни приложения
- Приложения за мобилни устройства
- XML уеб услуги
- Windows услуги
- Библиотеки с компоненти

Основните библиотеки, от които се състои FCL, са:

- Base Class Library – библиотека съдържаща основните средства, нужни за разработване на приложения. Дефинира работа с вход и изход, многозадачност, колекции, символни низове и интернационализация, достъп до мрежови ресурси, сигурност, отдалечено извикване и други.
- ADO.NET и XML – осигуряват достъп до бази данни и средства за обработка на XML.
- ASP.NET – предоставя ни рамкова среда (framework) за разработка на уеб приложения с богата функционалност, както и средства за създаване и консумиране на уеб услуги.
- Windows Forms – служи за основа при разработването на Windows приложения с прозоречно-базиран графичен потребителски интерфейс. Windows Forms се базира на вградените в Windows средства за изграждане на графичен потребителски интерфейс.

По-нататък ще обърнем специално внимание на всички тези библиотеки и ще разгледаме средствата, които те предлагат, в дълбочина.

Пакетите от FCL

За да се работи по-лесно с това голяма многообразие от типове, което FCL предлага, типовете са разделени в отделни асемблита и допълнително са разпределени в пространства от имена (namespaces) според своето предназначение и взаимовръзка.

Да разгледаме основните пространства от имена от FCL и тяхното предназначение:

- **`System`** – съдържа основни типове, използвани от всяко .NET приложение. В пространството **`System`** се намира, например, базовият за всички типове в .NET Framework клас **`System.Object`**, както и класът **`System.Console`**, който позволява вход и изход от конзолата.
- **`System.Collections`** – в това пространство се намират често използвани типове за управление на колекции от обекти: стек, опашка, хеш таблица и други.
- **`System.IO`** – съдържа типовете, които осигуряват входно-изходните операции в .NET Framework – потоци, ресурси от файловата система и други.
- **`System.Reflection`** – съдържа типове, които служат за достъп до метаданните по време на изпълнение на кода. Чрез тях е възможна реализацията на динамично зареждане и изпълнение на код.
- **`System.Runtime.Remoting`** – имплементира технология, която позволява отдалечен достъп до обекти и данни по прозрачен за програмиста начин.
- **`System.Runtime.Serialization`** – обединява типове, отговорни за процеса на сериализация и десериализация на обекти (запазване на състоянието на обект и по-късното му възстановяване).
- **`System.Security`** – в това пространство се намират типовете, които се използват за управление на сигурността. Те позволяват защита на данни и ресурси, определяне и проверка на текущите права на потребителя и други.
- **`System.Text`** – типовете от това пространство предоставят функционалност за обработка на текст, промяна на кодовата му таблица и други услуги, свързани с конвертиране на данни и интернационализация на приложенията.
- **`System.Threading`** – дефинира типове, осигуряващи достъп до нишки и свързаните с тях операции, като например синхронизация.
- **`System.Xml`** – съдържа типове за работа с XML и технологиите, свързани с него.

Освен тези общодостъпни пространства от имена, разполагаме и с още някои, които са достъпни за различните типове приложения:

- **`System.Web.Services`** – дефинира типовете, използвани за изграждането и консумирането на уеб услуги.
- **`System.Web.UI`** – съдържа стандартни средства и компоненти за изграждане на уеб приложения.
- **`System.Windows.Forms`** – съдържа типове, използвани при създаването на Windows приложения с графичен потребителски интерфейс.

Visual Studio .NET

До момента се запознахме с .NET Framework, с нейната архитектура, със средата за контролирано изпълнение на управляван код CLR и с основните библиотеки на .NET Framework.

Време е да разгледаме и средата за разработка на .NET приложения, която Microsoft предоставят на разработчиците. Това е продуктът **Microsoft Visual Studio .NET (VS.NET)**.

VS.NET е една от водещите в световен мащаб интегрирани среди за разработка на приложения (IDE – Integrated Development Environment). С негова помощ можем да извършваме всяка една от типичните задачи, свързани с изграждането на едно приложение – писане на код, създаване на потребителски интерфейс, компилиране, изпълняване и тестване, дебъгване, проследяване на грешките, създаване на инсталационни пакети, разглеждане на документацията и други.

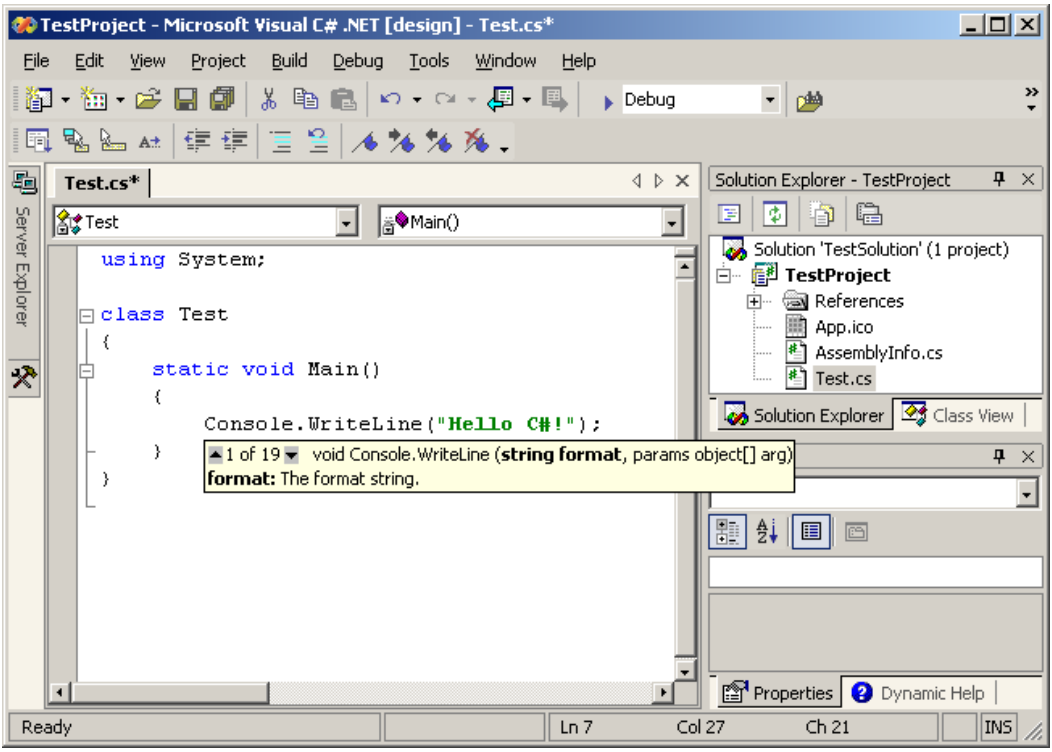
Писане на код

Пакетът Visual Studio .NET 2003 поддържа стандартно езиците за програмиране Microsoft C# .NET, Microsoft Visual Basic .NET, Microsoft C++ .NET (managed/unmanaged) и Microsoft Visual J#. За да ползвате език, различен от тези, които Microsoft предлага стандартно, трябва да инсталирате нужните добавки към VS.NET.

Текстовият редактор за код на VS.NET поддържа всички утвърдени съвременни функции на редакторите за сорс код – синтактично оцветяване за по-лесно визуално възприемане на кода и намаляване на грешките, автоматично довършване на започнат израз, автоматично извеждане на помощна информация по време на писане, средства за навигация по кода и много други.

Поддържа се IntelliSense функционалност за подсказване на имена на класове, методи и променливи. Тя предоставя огромно улеснение за навлизащите тепърва .NET програмисти, тъй като позволява те да разгледат на място възможностите и да изберат от списък тази, която ги интересува. Така се спестяват усилия, време за изписване на името и се намалява значително вероятността за досадни "правописни" грешки.

Следващата илюстрация дава нагледна представа за редактора на код на Visual Studio .NET 2003:



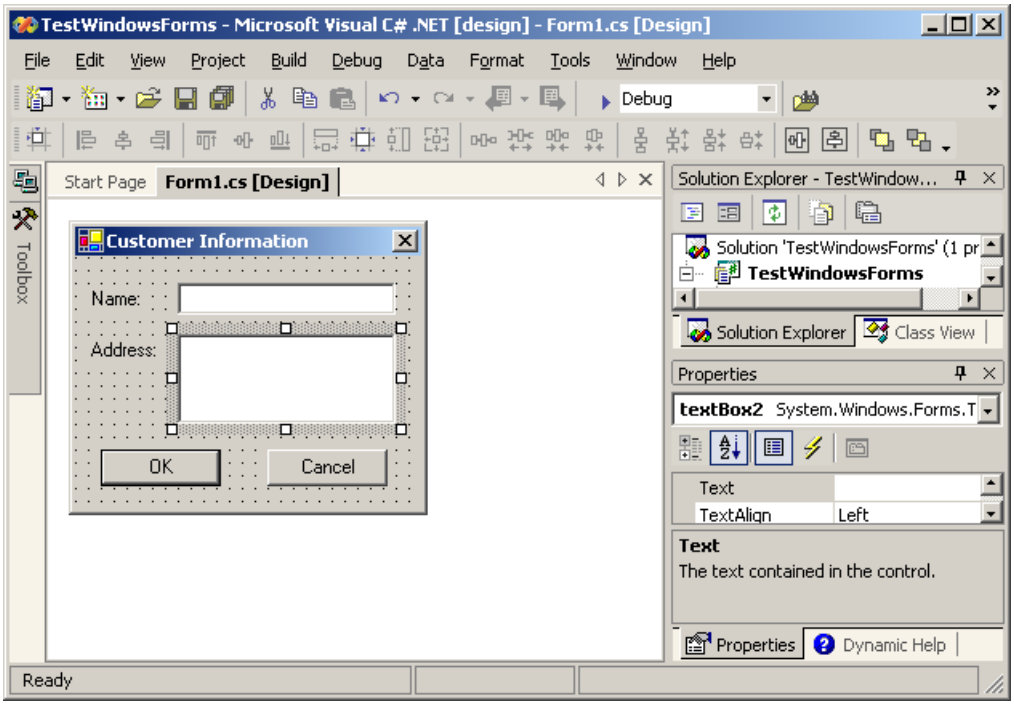
Създаване на потребителски интерфейс

Visual Studio .NET 2003 предоставя удобен за работа графичен дизайнер за потребителски интерфейси. С него за няколко минути можем да изградим дизайна на даден потребителски интерфейс, независимо дали става дума за Windows Forms прозорец, уеб страница или интерфейс за мобилни приложения.

Това, което виждаме във VS.NET докато изграждаме потребителския интерфейс, е почти същото, което и потребителят ще види, когато стартира приложението. Начинът на работа с различните технологии за представяне на потребителски интерфейс е много подобен и това допълнително улеснява разработчиците и повишава тяхната продуктивност.

Добавянето на визуални компоненти (или потребителски контроли) става чрез влачене и пускане (drag and drop), а след това ни остава само да настроим нужните свойства на обекта със желаните от нас стойности и да добавим обработчици към някои от събитията.

Ето изглед от VS.NET в момент на редактиране на диалог от Windows Forms приложение:



Компилиране

Visual Studio .NET 2003 предлага унифициран начин за работа с компилаторите за различните езици. Не се налага да използваме командния ред и да знаем дългия списък с инструкции към компилатора за да можем да компилираме кода и да създаваме асемблита. Достатъчно е на натиснем **`[Shift+Ctrl+B]`** за да компилираме цялото решение с всички проекти в него.

При компилация VS.NET автоматично създава нужните асемблита и ресурсни файлове. Тя се грижи и за сателитните асемблита (ако има такива), опреснява референциите към външни файлове и класове и изпълнява още много други задачи.

Освен синтактичните грешки, процесът на компилация улавя и някои семантични. Допълнително той може да показва и предупредителни съобщения за съмнителен или недобър код. Можем да контролираме нивото на филтриране на тези предупреждения и дори да настроим средата да ги счита за грешки и да прекъсва процеса на компилация заради тях.

Едно ограничение, което VS.NET има, е че то не може да създава многофайлови асемблита. Ако това наистина ни се наложи трябва да използваме инструмента Assembly Linker (**`al.exe`**).

VS.NET предлага два режима на компилация:

- **Debug** – в този режим компилаторът създава **дебъг символи** за всички методи в приложението и ги записва в отделен файл с разширение **.pdb**. Чрез него можем да извършваме проследяване на грешките (debugging). Този режим на компилация е препоръчителен за процеса на разработване и тестване на приложението.
- **Release** – в този режим на компилация VS.NET създава код, готов за продукция и разпространение до клиентите. От него са отстранени всички функции свързани с дебъгване и тестване. Не се генерират **.pdb** файлове и като цяло има по-добра производителност от **Debug** версията. За сметка на това възможностите за откриване грешки са намалени.

Изпълняване и тестване

Тъй като Visual Studio .NET интегрира в себе си разработването на приложения с различни технологии, ние можем да стартираме по унифициран начин всеки един от типовете приложения. Това, което трябва да направим, е единствено да натиснем бутона **Start** (или **Debug/Start**). Средата проверява дали има промени във файловете на проекта, ако има такива, тя прекомпилира приложението и след това стартира съответния процес.

VS.NET поддържа т. нар. решения (solutions). В едно решение може да има един или повече проекта. Например в една система може да 3 проекта – уеб услуга, Windows Forms клиент и ASP.NET уеб приложение.

Имаме възможност да указваме проект, който да се стартира при стартиране на решението. Допълнително можем да укажем да се стартират множество проекти при натискане на бутона **Start**. Тази опция е много удобна при разработване на клиент-сървър приложения, тъй като не ни се налага ръчно да стартираме всеки компонент.

Тестването на приложенията може да се извършва веднага след стартирането. Тъй като Visual Studio .NET 2003 се "закача" в дебъг режим към процеса на стартираното приложението, можем да дебъгваме лесно и бързо своя код.

Проследяване на грешки

Процесът на проследяване на грешки, или както по-често го наричаме дебъгване, се осъществява много лесно със Visual Studio .NET 2003. Средата ни предоставя множество вградени в нея инструменти, с които да извършваме тази задача. Инструментите са достъпни от менюто **Debug** и включват следните възможности:

- **Breakpoints** – списък със зададените точки на прекъсване (break points). Можем да премахваме, създаваме и настройваме параметрите на всяка точка поотделно.

- **Running documents** – списък с всички файлове, които се използват от приложението в момента. Използва се главно при дебъгване на уеб приложения.
- **Call stack** – показва ни стекът на извикванията на методите до дадения момент. Перфектен е за анализ на програмна логика и намиране на мястото, където е възникнало изключение.
- **Autos** – показва всички променливи, които са в момента в обхват.
- **Local** – показва всички локални променливи.
- **Immediate/Command Window** – позволява ни да изпълняваме инструкции и да променяме стойности на променливи по време на изпълнение. Предоставя множество мощни възможности, които обаче излизат извън рамките на нашата тема.
- **Watch** – показва списък с всички променливи, които сме заявили, че искаме да наблюдаваме. Чрез него можем и да променяме техните стойности по време на изпълнение на програмата.
- **Quick Watch** – показва стойността на избрана при дебъгване променлива.
- **Step control** – дава ни средства за постъпково изпълнение на кода ред по ред и стъпка по стъпка. Можем да избираме реда на изпълнение; да изпълняваме методи като влизаме в тях или ги изчакваме да завършват и преминаваме към следващата стъпка; можем да контролираме и кои редове от кода се изпълняват и при нужда да местим курсора на изпълнение напред и назад.
- **Exception control** – можем да задаваме дали нашето приложение да влиза в дебъг режим при възникване на изключение и да спира веднага след мястото на възникване на изключението без да сме сложили точка на прекъсване.

Освен тези удобства имаме възможност да разглеждаме съдържанието на паметта и регистрите в "суров" вид и да извършваме декомпилация на кода (disassembling).

Създаване на инсталационен пакет

Освен стандартните шаблони за всеки език за програмиране Visual Studio .NET 2003 ни предлага и шаблони за инсталационни пакети. Така се затваря цикълът на разработка на приложения. Можем да използваме готовите стандартни форми и технологии на инсталация и/или да добавим свои собствени към инсталационния пакет. След като сме завършили и тази стъпка, можем да разпространяваме своето приложение до крайните потребители.

Технологиите на инсталиране, които ни предлага Visual Studio .NET 2003, са приложими за почти всякакви приложения, независимо от това дали са конзолни, Windows Forms, уеб приложения или библиотеки с типове.

Процеса на създаване на инсталационни пакети е разгледан подробно в темата "Асемблита и разпространение".

Получаване на помощ

При инсталиране на Visual Studio .NET с него се инсталира неговата документация и по желание документацията на Microsoft .NET Framework (т.нар. MSDN Library).

Те автоматично се интегрират в средата за разработка и позволяват да получим т. нар. контекстно-ориентирана помощ. Например, докато използваме даден клас от .NET Framework, VS.NET ни показва неговата документация в прозореца **Dynamic Help**. Интегрираната помощна система във VS.NET позволява при натискане на клавиша **[F1]** да получим информация за текущия клас, свойство или компонент, който е на фокус. Това значително улеснява разработчика.

VS.NET е силно разширяема среда

Интегрираната среда за разработка Microsoft Visual Studio .NET е така проектирана, че лесно да може да се разширява с допълнителни модули и нови възможности. Съществуват стотици добавки (plug-ins) за VS.NET, които добавят поддръжка на нови езици и нови технологии, подпомагат процеса на разработка по различни начини, добавят интеграция с други продукти и т. н. Някои от тях са свободни, докато други са комерсиални продукти. Благодарение на добре документираните програмни интерфейси за интеграция с VS.NET програмистите могат да добавят и собствени добавки за средата.

Упражнения

1. Опишете накратко платформата Microsoft .NET. Кои са основните принципи, които са заложени в нея? Избройте четирите компонента, от които тя се състои.
2. Какво представляват .NET Enterprise сървърите? Избройте някои от тях. Какво представлява .NET Framework? От какви компоненти се състои? Какво е Visual Studio .NET? За какво служат .NET Building Block услугите? Какво са .NET Smart клиентите? Какво е характерно за тях?
3. Опишете накратко .NET Framework. От какви компоненти се състои?
4. Какво представлява средата за контролирано изпълнение на програмен код Common Language Runtime (CLR)?
5. Какво представлява Framework Class Library (FCL)? Каква функционалност предлага тя?
6. Какво е управляван код? Има ли причина да бъде използван вместо традиционния машиннозависим код? Какво е характерно за междинния език IL?

7. Какво представляват .NET асемблитата (assemblies)? Каква информация съдържат метаданните в асемблитата? Какво представляват .NET приложенията? Какво е област на приложението (application domain)?
8. Какво е Common Language Specification (CLS)? Защо е необходима тази спецификация? Какво описва тя?
9. Какво представлява общата система от типове в .NET Framework (Common Type System)? Защо е необходима тя?
10. Избройте няколко от .NET езиците. Какво е общото между тях? Какво е специфичното за всеки от тях?
11. Избройте основните пакети от Framework Class Library (FCL). За какво служат те?

Използвана литература

1. Светлин Наков, Архитектура на платформата .NET и .NET Framework – http://www.nakov.com/dotnet/lectures/Lecture-1-MS.NET-Framework-Architecture-v1.03.ppt
2. Jeffrey Richter, Applied Microsoft .NET Framework Programming, Microsoft Press, 2002, ISBN 0735614229
3. MSDN, Common Language Runtime Overview – http://msdn.microsoft.com/library/en-us/cpguide/html/cpconcommonlanguageruntimeoverview.asp
4. MSDN, Compiling to MSIL – http://msdn.microsoft.com/library/en-us/cpguide/html/cpconMicrosoftIntermediateLanguageMSIL.asp
5. MSDN, Application Domains Overview – http://msdn.microsoft.com/library/en-us/cpguide/html/cpconapplicationdomainsoverview.asp

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 2. Въведение в C#

Необходими знания

- Добро познаване на поне един език за програмиране от високо ниво (C, C++, Java, Pascal/Delphi, Perl, Python, PHP или друг)
- Базови познания за архитектурата на .NET Framework

Съдържание

- Принципи при дизайна на езика
- Нашата първа програма на C#
- Типове данни в C#. Примитивни типове данни. Изброен тип
- Декларации. Изрази. Оператори. Програмни конструкции
- Елементарни програмни конструкции. Съставни конструкции
- Конструкции за управление – условни конструкции, конструкции за цикъл, конструкции за преход. Специални конструкции
- Коментари в програмата
- Вход и изход от конзолата
- Дебъгерът на Visual Studio .NET
- XML документация в C# кода

В тази тема...

В настоящата тема ще разгледаме езика C#, ще се запознаем с неговите основни концепции, ще напишем и компилираме първата си C# програма. Ще се запознаем със средата за разработка Visual Studio .NET 2003 и ще демонстрираме работата с нейния дебъгер. Ще отделим внимание на типовете данни, изразите, програмните конструкции и конструкциите за управление в езика C#. Накрая ще демонстрираме колко лесно и полезно е XML документирането на кода в C#.

Настоящата тема има за цел да запознае читателя с конкретните синтактични правила на езика C# и неговите програмни конструкции без да претендира за изчерпателност. В нея няма да обясняваме какво е променлива, функция, цикъл и т. н., а ще се фокусираме върху реализацията на тези езикови примитиви в C#. Очаква се читателят да владее основите на програмирането с поне един език от високо ниво, а тази тема ще му помогне да премине към C#.

Какво е C#

C# е съвременен, обектно-ориентиран и типово обезопасен език за програмиране, който е наследник на C и C++. Той комбинира леснотата на използване на Java с мощността на C++.

Създаден от екипа на Андерс Хейлсбърг, архитектът на Delphi, C# заимства много от силните страни на Delphi – свойства, индексатори, компонентна ориентираност. C# въвежда и нови концепции – разделяне на типовете на два вида – стойностни (value types) и референтни (reference types), автоматично управление на паметта, делегати и събития, атрибути, XML документация и други. Той е стандартизиран от ECMA и ISO.

C# е специално проектиран за .NET Framework и е съобразен с неговите особености. Той е сравнително нов, съвременен език, който е заимствал силните страни на масово използваните езици за програмиране от високо ниво, като C, C++, Java, Delphi, PHP и др.

Принципи при дизайна на езика C#

Преди да се запознаем със синтаксиса и програмните конструкции в C#, нека първо разгледаме основните принципи, залегнали при проектирането му.

Компонентно-ориентиран

Езикът C# е насочен към компонентно-ориентираното програмиране, при което софтуерът се изгражда чрез съединяване на различни готови компоненти и описание на логиката на взаимодействие между тях.

При проектирането на .NET Framework и езика C# компонентният подход е залегнал на най-дълбоко архитектурно ниво. .NET Framework дефинира общ компонентен модел, който установява правилата за изграждане и използване на компоненти за всички .NET приложения. Езикът C# поддържа класове, интерфейси, свойства, събития и други средства за описание на компонентите, както и средства за тяхното използване. В темата "Графичен потребителски интерфейс с Windows Forms" ще дискутираме по-задълбочено компонентния модел на .NET Framework.

Всички данни са обекти

C# е обектно-ориентиран език за програмиране. В него залягат основните принципи на обектно-ориентираното програмиране, като капсулация на данните, наследяване и полиморфизъм.

В .NET Framework всички типове данни наследяват системния тип **`System.Object`** и придобиват от него някои общи методи, свойства и други характеристики.

В следствие на това в C# всички данни се третират като обекти. Дори примитивните типове, чрез въвеждането на автоматичното им опаковане (boxing) и разопаковане (unboxing) се превръщат в обекти. Например, **`5.ToString()`** е валидно извикване в C#, защото **`5`** се опакова и се разглежда като обект от тип **`System.Object`**, на който се извиква метода **`ToString()`**.

Сигурност и надеждност на кода

По идея .NET Framework и C# са проектирани за да осигурят висока сигурност и надеждност на изпълнявания софтуер. .NET Framework предоставя среда за контролирано изпълнение на управляван код, с което прави невъзможно възникването на някои от най-неприятните проблеми, свързани с управлението на паметта, неправилното преобразуване на типове и др. C# наследява всички тези характеристики от .NET Framework и добавя към тях някои допълнителни механизми за предпазване на програмистите от често срещани грешки.

Силна типизираност и типова безопасност

C# е силно типизиран и типово обезопасен. В него не се използват указатели към паметта, които създават много проблеми в по-старите езици за програмиране. Вместо тях се използват специални силно типизирани указатели, които се наричат **референции (references)**. Използването на референции вместо указатели решава проблемите, които възникват от неправилната работа с указатели и директния достъп до паметта. В .NET Framework управлението на паметта се извършва почти изцяло от CLR.

Всъщност в C# може да се използват указатели (като тези в C и C++) чрез запазената дума **`unsafe`**, но това не е се препоръчва в масовия случай, защото лишава програмата от типова обезопасеност и позволява неправилна работа с паметта.

В C# не може да се излезе от границите на масив или символен низ. При опит да бъде направено това, се получава изключение, което може да бъде прихванато и обработено. В езици като C, C++ и Pascal излизането от границите на масив води до достъп до памет, използвана от други данни и най-често пъти предизвиква сривове или неочаквано поведение на програмата.

При създаване на клас, структура или друг тип C# компилаторът не позволява да останат неинициализирани член-данни. Това защитава програмиста от възможността да работи с неинициализирани данни.

Макар C# да не инициализира автоматично локалните променливи, компилаторът предупреждава за неправилното им използване. Например следният код ще предизвика грешка при опит за компилация:

```
int value;
value = value + 5;
```

Преобразуването на типове също е безопасно. CLR не позволява да се извърши невалидно преобразуване на типове – да се преобразува променлива от даден тип към променлива от тип, който не е съвместим с първия. При опит да бъде направено това, възниква изключение.

Неявното преобразуване на типове е разрешено само за съвместими типове, когато не е възможна загуба на информация. При явно преобразуване на типове, ако те не са съвместими, се хвърля **InvalidCastException** по време на изпълнение. Например следният код предизвиква изключение по време на изпълнение:

```
object a = "This will raise InvalidCastException!";
int b = (int) a;
```

Безопасност на аритметичните операции

В C# чрез запазената дума **checked** могат да се отделят блокове код, в които аритметичните операции се проверяват за препълване на типовете и ако това се случи, се хвърля **OverflowException**. Това е много полезно, защото за разлика от C++, където при такива ситуации се получава грешен резултат, в C# може да се реагира адекватно на такава специфична ситуация. Ето един пример, при който CLR засича препълване на типа **int**:

```
checked
{
   int i = 100000;
   int j = i*i; // OverflowException is thrown
}
```

Експлицитно задаване на виртуални методи и припокриване на метод

Една от целите на езика C# е да позволи с малко усилия да се пише надежден код. С цел да се намалят грешките от припокриване на виртуални методи са въведени запазените думи **new** и **override**, чрез които да се контролира припокриването на виртуален метод, който е в базов клас при наследяване. Каква е разликата между двете? При полиморфизъм (обект от базовия клас е създаден като обект от наследника), ако е използвана **new** като модификатор на метода в наследника, ще се извика функцията на базовия клас, а при използване на **override** – функцията на наследника.

Автоматично управление на паметта и ресурсите

В .NET Framework заделянето и използването на паметта се управлява автоматично от CLR (Common Language Runtime). Стойностните типове, които ще разгледаме по-подробно в темата "Обща система от типове", се пазят в стека, докато референтните – в т. нар. "динамична памет"

(managed heap), за която се грижи системата за почистване на паметта (garbage collector).

Системата за почистване на паметта е част от CLR и нейна задача е да освобождава периодично паметта и ресурсите, заделени за обекти, които не се използват повече от приложението. Такива обекти могат да бъдат най-разнообразни: данни в динамичната памет, масиви, символни низове, а също и файлове, буфери в паметта, връзки към бази данни и др.

Грижата за паметта е трудна и сложна задача, но благодарение на CLR тя не е задължение на .NET програмистите. На нея ще обърнем специално внимание в темата "Управление на паметта и ресурсите".

Широко използване на изключения

Обработката на грешки, които могат да възникнат по време на изпълнение на програмата, в .NET Framework се реализира чрез използване на изключения. Механизмът на изключенията позволява да се съобщи за възникнал проблем или неочаквана ситуация и за нея да може да се реагира адекватно.

Изключенията представляват обекти от клас **`Exception`** или производен на него клас и съдържат информация за възникналата грешка. Например, при опит за деление на нула CLR прихваща проблема и предизвиква изключението **`DivideByZeroException`**, а при опит за излизане от границите на масив възниква **`ArgumentOutOfRangeException`**. Работата с изключения също ще бъде дискутирана в детайли в темата "Управление на изключенията в .NET".

Вградени механизми за сигурност на кода

В .NET Framework са въведени т. нар. сигурност на ниво достъп до кода (code access security) и сигурност, базирана на роли (role-based security). Чрез тях се осъществява контрол на достъпа до ресурси от програмата. Например, ако трябва да се извика системна функция или да се пише във файл, кодът трябва да има права да го направи. Сигурността на ниво достъп до кода оставя CLR да взима решения, докато при сигурност, базирана на роли, програмата може да реагира различно спрямо ролята и правата на потребителя.

Всичкият код е на едно място

В C# няма разделяне на хедър файлове и файлове с имплементация, както в C и C++. Това спестява много проблеми и улеснява поддръжката на сорс кода. В C# всичкият програмен код на даден клас е в един файл.

Програмите на C#

Програмите на C# представляват съвкупност от дефиниции на класове, структури и други типове. Във всяка C# програма някой от класовете съдържа метод **Main()** – входна точка за програмата.

Приложенията могат да се състоят от много файлове, а в един файл може да има няколко класове, структури и други типове.

Класовете логически се разполагат в **пространства от имена** (**namespaces**). Те от своя страна могат да се систематизират в йерархия, т.е. едно пространство от имена може да съдържа както класове, така и други пространства от имена. Едно пространство от имена може да е разположено в няколко файла и дори в няколко асемблита. Например, в пространството от имена **System** се съдържа пространството от имена **Xml**. Пространството **System.Xml** от своя страна е разделено в две различни асемблита - **System.Xml.dll** и **System.Data.dll**.

Във Visual Studio .NET има инструмент, наречен "Object Browser", чрез който могат да се разгледат йерархиите на пространствата от имена в проекта, какво съдържат те и в кои файлове се намират.

Нашата първа програма на C#

Всяка книга за запознаване с даден програмен език започва обикновено с програмката "Hello, world!". Ние няма да правим изключение от този принцип и ще започнем по подобен начин – с програмката "Hello, C#". Ето как изглежда нейният сорс код:

HelloCSharp.cs

```
using System;

class HelloCSharp
{
   static void Main()
   {
      Console.WriteLine("Hello, C#");
   }
}
```

Как работи програмата?

На първия ред директивата **using System** указва, че се използва пространството от имена **System** (т.е. всички класове, структури и други типове, декларирани в него). Тя е като **#include** в C++, като **import** в Java и като **uses** в Delphi.

Следва декларацията на клас с ключова дума **class**. Този клас се състои от един единствен метод – методът **static void Main()**, който е входна точка на програмата. Когато този метод завърши, завършва и програмата.

В метода **Main()** се извиква метода **WriteLine(…)** на класа **Console**, намиращ се в пространството от имена **System**. Класът **Console** осигурява средства за вход и изход от конзолата. Чрез него отпечатваме на конзолата текста "Hello, C#".

Компилиране от командния ред

Програми на C# могат да се компилират от командния ред чрез компилатора **csc.exe**, който е стандартна част от .NET Framework и е достъпен от директорията, в която е инсталиран той. При стандартна инсталация тя е **C:\Windows\Microsoft.NET\Framework\v1.1.4322**.

Можем да компилираме сорс кода на примерната програма по следния начин: Използвайки командния интерпретатор (**cmd.exe**) се придвижваме до директорията, където се намира файлът **HelloCSharp.cs**. След това можем да го компилираме със следната команда:

```
csc HelloCSharp.cs
```

За да бъде намерен компилаторът **csc.exe**, е необходимо в текущия път (в променливата **PATH** от средата) да е включена директорията на .NET Framework.

Ако компилацията премине успешно, в резултат се получава файлът **HelloCSharp.exe**, който представлява .NET асембли, записано като изпълним файл.

Стартиране от командния ред

Стартирането на получения изпълним (**.exe**) файл става както всички останали изпълними файлове, например чрез следната команда:

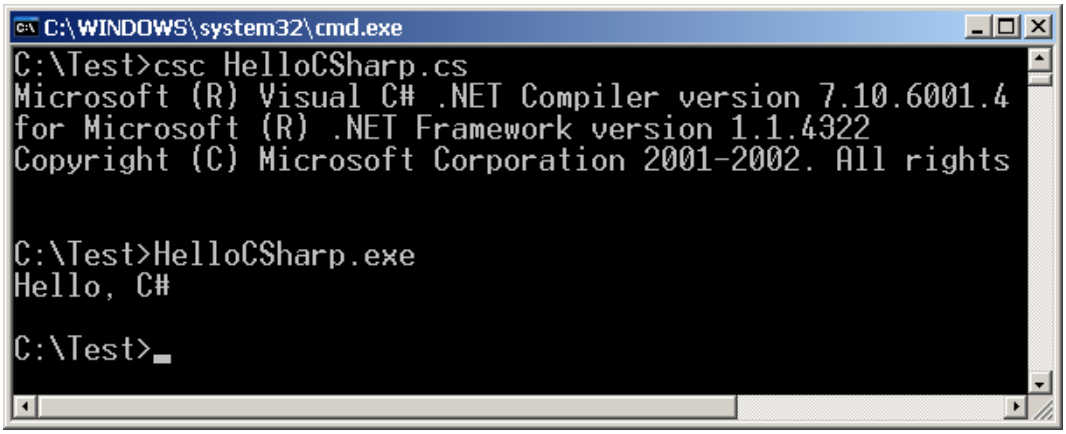
```
HelloCSharp.exe
```

Резултатът

Резултатът от изпълнението на нашата първа C# програмка представлява един текстов ред:

```
Hello, C#
```

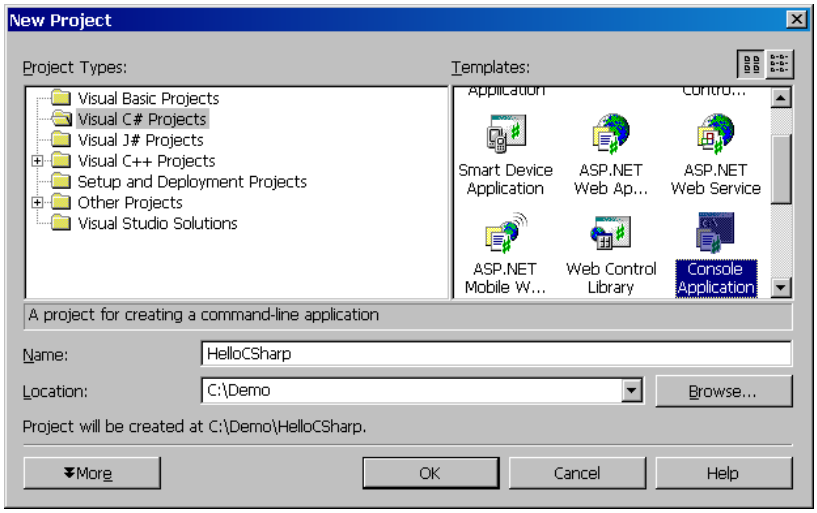
Резултатът от компилирането и изпълнението на примерната програма е показан на следващата картинка:



Създаване на проект, компилиране и стартиране от Visual Studio.NET

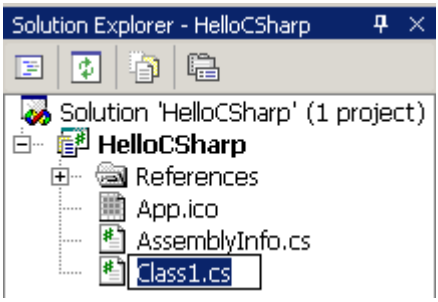
Ще покажем как може да се използва интегрираната среда за разработка на приложения Microsoft Visual Studio .NET за изпълнение на предходната примерна програмка. Ще създадем нов проект (конзолно приложение), ще го компилираме и изпълним. Трябва да преминем през следните стъпки:

1. Стартираме Visual Studio .NET.
2. От меню **File** избираме **New Project**. Избираме **Visual C# Projects | Console Application**. Избираме име и местоположение за проекта:

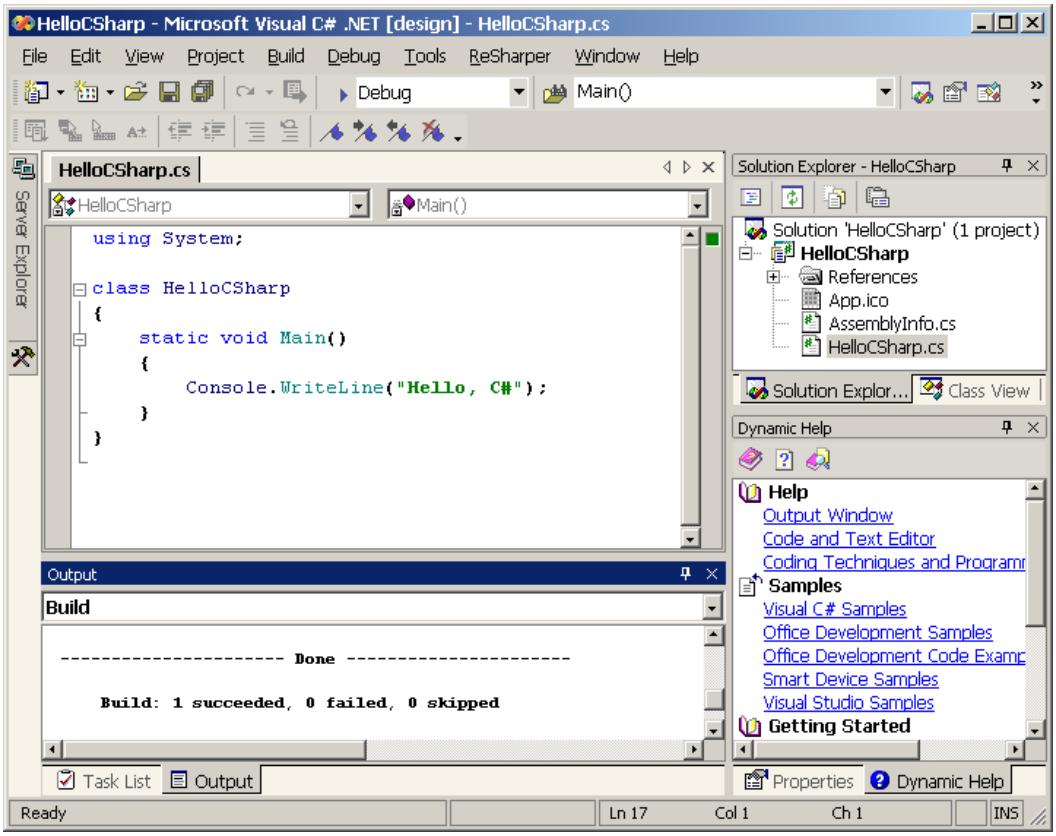


Visual Studio .NET създава за нас един Solution и един проект в него, съдържащ няколко файла. Файлът, в който можем да пишем нашия код, се отваря автоматично.

3. Въвеждаме примерната програмка. Можем да сменим името на файла `Class1.cs` с `HelloCSharp.cs` чрез клавиша [F2], натиснат в момент, в който е активен файлът `Class1.cs` от Solution Explorer:



4. За да компилираме, натискаме [Shift+Ctrl+B] или избираме менюто **Build | Build Solution**. Ето как изглежда VS.NET в този момент:



5. За да стартираме приложението, натискаме [Ctrl+F5] или избираме от менюто **Debug | Start Without Debugging**. В резултат приложението се изпълнява в нов конзолен прозорец и след приключване на работата му VS.NET ни приканва да натиснем някакъв клавиш, за да затвори прозореца:

Можем да стартираме приложението и само с [F5], но тогава то ще се изпълни в режим на дебъгване и след приключване на работата му прозорецът, в който е изведен резултата, веднага ще се затвори и няма да го видим.

Запазени думи в C#

Езикът C# дефинира следните запазени думи, които се използват в конструкциите и синтаксиса на езика:

abstract	as	base	bool	break	byte
case	catch	char	checked	class	const
continue	decimal	default	delegate	do	double
else	enum	event	explicit	extern	false
finally	fixed	float	for	foreach	goto
if	implicit	in	int	interface	internal
is	lock	long	namespace	new	null
object	operator	out	override	params	private
protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	string
struct	switch	this	throw	true	try
typeof	uint	ulong	unchecked	unsafe	ushort
using	virtual	void	volatile	while	

Ще видим за какво служат повечето от тях постепенно, в процеса на запознаване с езика C#, с обектно-ориентираното програмиране в .NET Framework, с общата система от типове и в някои други теми.

Типове данни в C#

Типовете данни в C# биват два вида – типове по стойност (value types) и типове по референция (reference types). Типовете по стойност (стойностни типове) директно съдържат своята стойност и се съхраняват в стека. Те се предават по стойност. Типовете по референция (референтни типове) представляват силно типизирани указатели към стойност в динамичната

памет. Те се предават по референция (адрес) и се унищожават от **garbage collector**, когато не се използват повече от програмата.

Типовете биват още примитивни (вградени, built-in) типове и типове, дефинирани от потребителя.

Стойностни типове (value types)

Типовете по стойност (стойностни типове) са примитивните типове, изброените типове и структурите. Например:

```
int i; // примитивен тип int
enum State { Off, On } // изброен тип (enum)
struct Point { int x, y; } // структура (struct)
```

Референтни типове (reference types)

Типовете по референция (референтни типове) са класовете, интерфейсите, масивите и делегатите. Например:

```
class Foo: Bar, IFoo {...} // клас
interface IFoo: IBar {...} // интерфейс
string[] a = new string[5]; // масив
delegate void Empty(); // делегат
```

На всички типове в C# съответстват типове от общата система от типове (Common Type System – CTS) на .NET Framework. Например, на примитивния C# тип **`int`** съответства типа **`System.Int32`** от CTS.

Примитивни типове

Примитивните типове данни в C# (built-in data types) биват:

Примитивни типове по стойност

- **`byte`**, **`sbyte`**, **`int`**, **`uint`**, **`long`**, **`ulong`** – цели числа
- **`float`**, **`double`**, **`decimal`** – реални числа
- **`char`** – Unicode символи
- **`bool`** – булев тип (**`true`** или **`false`**)

Примитивни типове по референция

- **`string`** – символен низ (неизменима последователност от Unicode символи)
- **`object`** – обект (специален тип, който се наследява от всички типове)

Типове дефинирани от потребителя

Типовете дефинирани от потребителя биват класове, структури, изброени типове, интерфейси и делегати:

```
class Foo: Bar, IFoo {...} // клас
struct Point { int x, y; } // структура
interface IFoo: IBar {...} // интерфейс
delegate void Empty(); // делегат
```

Вече се сблъскахме с класовете в C# в примерната програма "Hello, C#". Повече за тях, както и за структурите и интерфейсите ще научим в темата "Обектно-ориентирано програмиране в .NET".

Преобразуване на типовете

Има два типа преобразувания на примитивните типове – преобразуване по подразбиране (implicit conversion) и изрично преобразуване (explicit conversion).

В C# преобразуването по подразбиране е позволено, когато е безопасно. Например, от **int** към **long**, от **float** към **double**, от **byte** към **short**:

```
short a = 10;
int b = a; // implicit type conversion from short to int
```

Изричното преобразуване се използва, когато преобразуваме към по-малък тип или типовете не са директно съвместими. Например, от **long** към **int**, от **double** към **float**, от **char** към **short**, от **int** към **char**, от **sbyte** към **uint**:

```
int a = 10;
short b = (short) a; // explicit type conversion
```

В C# има специална ключова дума **checked**, която указва при препълване да се получава **System.OverflowException** вместо грешен резултат. Ключовата дума **unchecked** действа противоположна на **checked**. Ето пример за преобразуване на типове с използване на тези ключови думи:

```
byte b8 = 255;
short sh16 = b8; // implicit conversion
int i32 = sh16; // implicit conversion
float f = i32; // implicit - possible loss of precision!
double d = f; // implicit conversion

checked
{
  byte byte8 = (byte) sh16; // explicit conversion
  // OverflowException is possible!
```

```
   ushort ush16 = (ushort) sh16; // explicit conversion
   // OverflowException is possible if sh16 is negative!
}

unchecked
{
   uint ui32 = 1234567890;
   sbyte sb8 = (sbyte) ui32; // explicit conversion
   // OverflowException is not thrown in unchecked mode
}
```

Изброени типове (enumerations)

Изброените типове в C# се състоят от множество именувани константи. Дефинират се със запазената дума **enum** и наследяват типа **System.Enum**. Ето пример за изброен тип, който съответства на дните от седмицата:

```
public enum Days
{
   Saturday,
   Sunday,
   Monday,
   Tuesday,
   Wednesday,
   Thursday,
   Friday
};
```

Изброените типове се използват за задаване на една измежду няколко възможности. Вътрешно се представят с **int**, но може да се зададе и друг числов тип.

Изброените типове са силно типизирани – те не се превръщат в **int**, освен експлицитно.

Ето пример как може да бъде използван даден изброен тип:

```
Days today = Days.Friday;
if (today == Days.Friday)
{
   Console.WriteLine("Днес е петък.");
}
```

Както се вижда, инстанциите на изброените типове могат да приемат една от дефинираните в тях стойности.

Изброените типове могат да се използват и като съвкупност от битови флагове чрез атрибута **[Flags]**. Ето пример за изброен тип, който може да приема за стойност комбинация от дефинираните в него константи:

```
[Flags]
public enum FileAccess
{
   Read = 1,
   Write = 2,
   Execute = 4,
   ReadWrite = Read | Write
}

// ...

Console.WriteLine(
   FileAccess.ReadWrite | FileAccess.Execute);

// The result is: "ReadWrite, Execute"
```

Какво представляват атрибутите и как се използват ще разгледаме по-детайлно в темата "Атрибути". Засега трябва да знаем, че чрез тях може да се асоциира допълнителна информация към типовете.

Използването на изброени типове осигурява по-високо ниво на абстракция и по този начин сорс кодът става по-разбираем и по-лесен за поддръжка.

В .NET Framework широко се използват изброени типове. Например, изброения тип **`ConnectionState`**, намиращ се в пространство от имена **`System.Data`**, характеризира състоянието на връзка към база от данни, създадена чрез ADO.NET (зададено е и числовото съответствие на всяко едно от състоянията):

```
public enum ConnectionState
{
   Closed = 0,
   Open = 1,
   Connecting = 2,
   Executing = 4,
   Fetching = 8,
   Broken = 16
}
```

Идентификатори

Идентификаторите в C# се състоят от последователности от букви, цифри и знак за подчертаване като винаги започват с буква или знак за подчертаване. В тях малките и главните букви се различават. Идентификаторите могат да съдържат Unicode символи, например:

```
int алабала_портокала = 42;
bool \u1027\u11af = true;
```

Microsoft препоръчва се следната конвенция за именуване:

- PascalCase – за имена на класове, пространства от имена, структури, типове, методи, свойства, константи
- camelCase – за имена на променливи и параметри

Въпреки, че е възможно, не се препоръчва да се използват идентификатори на кирилица или друга азбука, различна от латинската.

Декларации

Декларациите на променливи в C# могат да са няколко вида (почти като в C++, Java и Delphi) – локални променливи (за даден блок), член-променливи на типа и константи. Ето пример:

```
int count;
string message;
```

Член-променливите могат да имат модификатори, например:

```
public static int mCounter;
```

Константи

Константите в C# биват два вида – константи, които приемат стойността си по време на компилация (compile-time константи) и такива, които получават стойност по време на изпълнение на програмата (runtime константи).

Compile-time константи

Compile-time константите се декларират със запазената дума **`const`**. Те задължително се инициализират в момента на деклариране им и не могат да се променят след това. Те реално не съществуват като променливи в програмата. По време на компилация се заместват със стойността им. Например:

```
public const double PI = 3.1415926535897932;
const string COMPANY_NAME = "Менте Софт";
```

Runtime константи

Runtime константите се декларират като полета с модификатора **`readonly`**. Представляват полета на типа, които са само за четене. Инициализират се по време на изпълнение (в момента на деклариране или в конструктора на типа) и не могат да се променят след като веднъж са инициализирани. Например:

```
public readonly DateTime NOW = DateTime.Now;
```

Оператори

Операторите в C# са много близки до операторите в C++ и Java и имат същите действие и приоритет. Те биват:

- Аритметични: **+, -, *, /, %, ++, --**
- Логически: **&&, ||, !, ^, true, false**
- Побитови операции: **&, |, ^, ~, <<, >>**
- За слепване на символни низове: **+**
- За сравнение: **==, !=, <, >, <=, >=**
- За присвояване: **=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=**
- За работа с типове: **as**, **is**, **sizeof**, **typeof**
- Други: **., [], (), ?:, new, checked, unchecked, unsafe**

В C# операторите могат да се предефинират. В темата "Обектно-ориентирано програмиране в .NET" ще видим как точно става това.

Изрази (expressions)

Програмен код, който се изчислява до някаква стойност, се нарича израз (expression). Изразите в C# имат синтаксиса на C++ и Java. Например:

```
a = b = c = 20; // израз със стойност 20
(a+5)*(32-a)%b  // израз с числова стойност
"ала" + "бала" // символен израз (string)
Math.Cos(Math.PI/x) // израз с реална стойност
typeof(obj) // израз от тип System.Type
(int) arr[idx1][idx2] // израз от тип int
new Student() // израз от тип Student
(currentValue <= MAX_VALUE) // булев израз
```

Програмни конструкции (statements)

Програмните конструкции (statements) имат синтаксиса на C++ и Java. Те биват няколко вида:

Елементарни програмни конструкции

Елементарните програмни конструкции са най-простите елементи на програмата. Например:

```
// присвояване (<променлива> = <израз>)
sum = (a+b)/2;

// извикване на метод
```

```
PrintReport(report);

// създаване на обект
student = new Student("Светлин Наков", 3, 42688);
```

Съставни конструкции

Съставните програмни конструкции се състоят от няколко други конструкции, оградени в блок. Например:

```
{
   Report report = GenerateReport(period);
   report.Print();
}
```

Програмни конструкции за управление

Конструкциите за управление, както в повечето езици за програмиране, биват условни конструкции, конструкции за цикъл, за преход и т. н. В C# синтаксисът на тези конструкции е много близък до синтаксиса на C++ и Java.

Условни конструкции (conditional statements)

Условните конструкции в C# са **if**, **if-else** и **switch**. Техният синтаксис е еднакъв със синтаксиса им в C, C++ и Java.

if и **if-else** конструкциите за разлика от C и C++ могат да приемат единствено булево условие. Не са позволени целочислени стойности, които да играят ролята на **true**, ако са различни от **0** и **false** – иначе. Ето няколко примера за условна конструкция:

```
if (orderItem.Ammount > ammountInStock)
{
   MessageBox.Show("Not in stock!", "error");
}

if (Valid(order))
{
   ProcessOrder(order);
}
else
{
   MessageBox.Show("Invalid order!", "error");
}
```

Ново в **switch** конструкцията за разлика от C и C++ е, че позволява изразът, по който се осъществява условието, да бъде от тип **string** или **enum**. Например:

```
switch (characterCase)
{
   case CharacterCasing.Lower:
      text = text.ToLower();
      break;
   case CharacterCasing.Upper:
      text = text.ToUpper();
      break;
   default:
      MessageBox.Show("Invalid case!", "error");
      break;
}
```

Конструкцията **switch** се различава от реализацията си в C++. В C# не се разрешава "пропадане" (fall-through). Пропадането в **switch** конструкциите може да доведе до грешки. Независимо от удобствата, които предлага тази възможност, дизайнерите на езика C# са преценили, че рискът за грешка поради пропускане на **break** е по-голям, затова всеки **case** етикет трябва задължително да завършва с **break**.

Конструкции за повторение и цикъл

Конструкциите за повторение (iteration statements) са **for**-цикъл, **while**-цикъл, цикъл **do-while** и цикъл **foreach** – за обработка на колекции. Техният синтаксис е еднакъв със синтаксиса им в C, C++ и Java. Изключение прави **foreach** цикълът, който няма еквивалент в C и C++. Ето няколко примера:

Пример за **for**-цикъл:

```
// Отпечатваме числата от 1 до 100 и техните квадрати
for (int i=1; i<=100; i++)
{
   int i2 = i*i;
   Console.WriteLine(i + " * " + i + " = " + i2);
}
```

Пример за **while**-цикъл:

```
// Изчисляваме result = a^b
result = 1;
while (b > 0)
{
   result = result * a;
   b--;
}
```

Пример за цикъл **do-while**:

```
// Четем символи до достигане на край на ред
do
{
   ch = ReadNextCharacter(stream);
}
while (ch != '\n');
```

Операторът **foreach** е приложим за масиви, колекции и други типове, които поддържат интерфейса **IEnumerable** или имат метод за извличане на итератор (enumerator).

Пример за цикъл **foreach**:

```
string[] names = GetNames();

// Отпечатваме всички елементи на масива names
foreach (string name in names)
{
   Console.WriteLine(name);
}
```

Конструкции за преход

Конструкциите за преход в C# са: **break**, **continue** – които се използват в цикли, **goto** – за безусловен преход и **return** – за връщане от метод. Те работят по същия начин, като в C, C++ и Java.

Пример за използване на конструкцията **break**:

```
// Търсим позицията на даден елемент target в масива a[]
int position = -1;
for (int i=0; i<a.length; i++)
{
   if (a[i] == target)
   {
      position = i;
      break;
   }
}
return position;
```

Конструкции за управление на изключенията

Конструкциите за управление на изключенията в C# са: **throw** – за предизвикване на изключение, **try-catch** – за прихващане на изключение, **try-finally** – за сигурно изпълнение на завършваща секция и **try-catch-finally** – за прихващане на изключение със завършваща секция.

Пример за предизвикване и прихващане на изключение:

```
// ...
public static void ThrowException()
{
   // ...
   throw new System.Exception();
}
// ...
public static void Main()
{
   try
   {
      ThrowException();
   }
   catch(System.Exception e)
   {
      // ...
   }
   finally
   {
      // ...
   }
}
```

Методът **ThrowException()** предизвиква изключение от тип **Exception**, което може да бъде прихванато, ако функцията, която го предизвиква, се намира в **try-catch** блок. В такъв случай може да се извършат действия в **catch** блока, в зависимост от информацията, която носи това изключение.

Конструкциите **try-finally** и **try-catch-finally** се използват най-вече за освобождаване на ресурси, които се използват в тялото им. Независимо дали възникне изключение при работата с даден ресурс, той трябва да бъде освободен накрая – това обикновено се прави във **finally** блок, който се изпълнява независимо дали се минава през **catch** блока. Блокът **finally** се изпълнява дори и да има **return** в **catch** или **try** блок.

В темата "Управление на изключенията в .NET" ще разгледаме подробно работата с изключения, техните особености и препоръките за правилна работа с тях.

Специални конструкции

Специалните конструкции в C# са: **lock** – за синхронизирано изпълнение, **checked**, **unchecked** – за контрол на аритметичните препълвания, **unsafe** – за директен достъп до паметта чрез указатели, **fixed** – за фиксиране на местоположението в паметта при работа с неуправляван код.

Коментари в програмата

Коментарите биват два вида - коментар за част от програмен ред и блоков коментар.

Ето пример за коментар на един ред:

```
Order orders[]; // Съдържа всички поръчки на потребителя
```

Ето пример и за блоков коментар:

```
/* Изтриваме всички поръчки, за които някой артикул не
е наличен в необходимото количество. Изтриване реално
не се извършва, а само се променя статуса на Canceled */
foreach (Order order in customer.Orders)
{
   if (!AllItemsInStock(order))
   {
      order.Status = OrderStatus.Canceled;
   }
}
```

Вход и изход от конзолата

В .NET Framework за вход и изход от конзолата се използват стандартни класове от BCL (Base Class Library). Входът и изходът от конзолата се осъществяват чрез класа **Console**, намиращ се в пространство от имена **System**.

Класът **System.Console** предоставя основната функционалност, от която се нуждаят конзолните приложения (console applications), които четат и пишат на екрана. Ако конзолата не съществува (например в Windows Forms и уеб-базираните приложения), писането в конзолата няма никакъв ефект и не се предизвикват изключения.

Всяко конзолно приложение при стартиране получава от операционната система три стандартни потока – за вход, изход и за грешки. При изпълнение от конзолата тези три потока автоматично се асоциират със самата нея. За достъп до стандартния вход, стандартния изход и стандартния изход за грешки в .NET Framework се използват свойствата **In**, **Out** и **Error** на класа **Console**.

Вход от конзолата

Входът от конзолата се осъществява чрез два метода на класа **Console** – **Read()** и **ReadLine()**. Методът **Read()** чете единичен символ от стандартния вход и го връща като **int** стойност или връща **-1**, ако няма повече символи. Методът **ReadLine()** чете цял символен ред и връща **string** или стойност **null** ако е достигнат края на входа.

И двата метода са синхронни (блокиращи) операции т. е. при извикване блокират, докато не бъде прочетен някакъв символ (ред).

Методът **Read()** има една особеност – той не връща управлението след всеки въведен символ, а връща прочетените символи наведнъж един след друг едва след като се натисне **[Enter]**. По тази причина този метод не е удобен за интерактивен вход от клавиатурата при конзолни приложения.

Вход от конзолата – примери

Ето един пример за използването на метода **Read()**:

```
while (true)
{
   int i = Console.Read();
   if (i == -1)
   {
      break;
   }
   char c = (char) i;
   Console.WriteLine ("Echo: {0}", c);
}
```

В практиката за въвеждане на стойности от конзолата по-често се използва методът **ReadLine()**. Ето пример за неговата употреба:

```
string s = Console.ReadLine();
Console.WriteLine("You enetered: {0}", s);
```

Изход към конзолата

Изходът към конзолата се осъществява чрез два метода на класа **Console** - **Write(…)** и **WriteLine(…)**, които печатат на конзолата подадените като параметри данни, с разликата, че **WriteLine(…)** преминава на нов ред след като отпечата текста. Методите приемат **string**, **int**, **float**, **double** и други типове данни.

Write(…) и **WriteLine(…)** приемат и параметрични форматиращи низове, които позволяват печатане на текст чрез шаблони, които се попълват от подадените параметри. На форматиращите низове ще обърнем специално внимание в темата "Символни низове".

Изход на конзолата – пример

Ето един пример за вход и изход от конзолата, който илюстрира и използването на форматиращи низове:

```
int a = Int32.Parse(Console.ReadLine());
int b = Int32.Parse(Console.ReadLine());
Console.WriteLine("{0} + {1} = {2}", a, b, a+b);
```

```
// (въвеждаме съответно 2 и 3 като вход от конзолата)
// Резултат: 2 + 3 = 5

Console.WriteLine(
   "Днес е {0:dd.MM.yyyy} г.", DateTime.Now);
// Резултат: Днес е 13.05.2004 г.

Console.WriteLine("Цена: {0,12:C}", 27);
// Резултат: Цена:     27,00 лв
// (точният формат зависи от текущите езикови настройки)

string name = Console.ReadLine();
Console.WriteLine("Хей, {0}, часът е {1:HH:mm}!",
   name, DateTime.Now);
// (въвеждаме "Наков")
// Резултат: Хей, Наков, часът е 16:43!
```

Дебъгерът на Visual Studio .NET

Сега ще илюстрираме как се използва дебъгерът на Visual Studio .NET. Ще покажем поставяне на точка за спиране (breakpoint), изпълнение на програмата в дебъг режим, проследяване на изпълнението на програмата и следене на стойностите на променливите по време на изпълнение.

Ще си поставим за задача да напишем програма, която намира всички трицифрени числа, сумата от цифрите на които е стойността 25. Можем да решим задачата по следния начин:

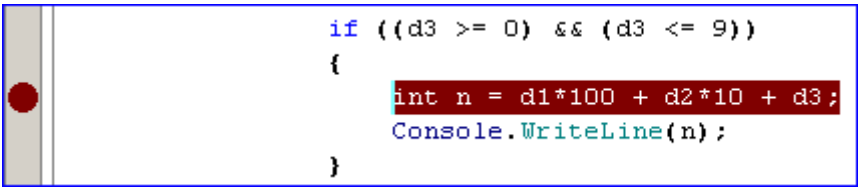
Digits.cs

```
using System;

public class Digits
{
   static void Main()
   {
      for (int d1=1; d1<=9; d1++)
      {
         for (int d2=0; d2<=9; d2++)
         {
            int d3 = 25 - d1 - d2;
            if ((d3 >= 0) && (d3 <= 9))
            {
               int n = d1*100 + d2*10 + d3;
               Console.WriteLine(n);
            }
         }
      }
   }
```

```
}
```

За да проследим изпълнението на примерната програма, ще изпълним следните стъпки:

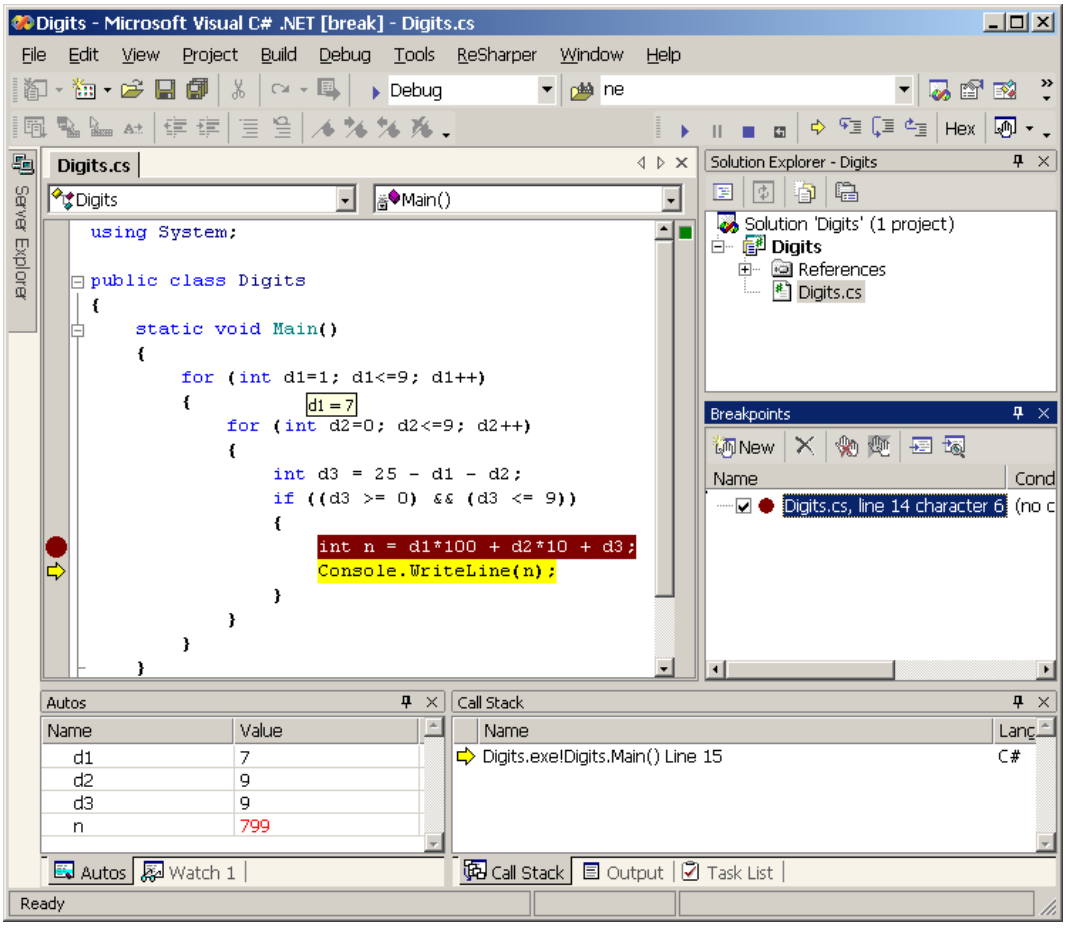
1. Стартираме Visual Studio .NET.
2. Създаваме ново конзолно приложение с име **`Digits.sln`**. Въвеждаме в него сорс кода от примера **`Digist.cs`**.
3. За да го компилираме натискаме [Shift]+[Ctrl]+[B].
4. Слагаме точка на прекъсване (breakpoint) с мишката върху първия ред от най-вътрешния програмен блок (щракваме малко вляво от самия ред и редът се маркира по специфичен начин):



5. Стартираме програмата от съответния бутон за стартиране от лентата с инструменти на Visual Studio .NET. Програмата ще започне да се изпълнява и когато се достигне реда с точката на прекъсване, Visual Studio .NET ще спре изпълнението и ще влезе в дебъг режим.
6. От менюто **Debug** можем да разгледаме по-интересните функции на дебъгера на Visual Studio .NET. Можем да добавим точки на прекъсване (breakpoints), да следим стойностите на променливите, да проследяване на изпълнението на кода по различен начин (**Step Into**, **Step over**, **Step out**), да добавим променливи за следене (**Add Watch**), да следим стека за изпълнение на програмата и т.н. Можем да разгледаме представянето на променливите в паметта – като изберем **Debug | Windows | Memory**. Много полезен е и прозорецът **Command Window**, в който не само може да се види стойност на променлива в дебъг режим, но и да се изпълни някакъв метод и да се види върнатата от него стойност. С [Shift] + [F9] при маркирана променлива може да се извика прозорец за нейното наблюдение (**Quick Watch**).

Точките на прекъсване могат да бъдат асоциирани с някакво условие (conditional breakpoints) и да спират изпълнението на програмата само ако това условие е истина.

Ето как изглежда работното пространство на Visual Studio .NET по време на проследяване на изпълнението на програмата след спиране в точката на прекъсване и преминаване към следващия оператор с [F10]:



Инструментът ILDASM

Всяка програма на C# се компилира до междинен език IL (Intermediate Language). Microsoft предоставя стандартен инструмент за разглеждане на този, генериран от компилаторите на C#, код. Това е инструментът **Microsoft .NET Framework Disassembler (ILDASM)**. С тази деасемблираща програма можем да отворим всяко .NET асембли и да разгледаме неговите пространства от имена, класове, типове и код.

Инструментът **`ildasm.exe`** е стандартна част от Microsoft .NET Framework SDK. Обикновено .NET Framework SDK идва заедно с Visual Studio .NET и се намира в директория **`C:\Program Files\Microsoft Visual Studio .NET 2003\SDK\v1.1\Bin`**. Нека илюстрираме как се използва той. За целта трябва да изпълним следните стъпки:

1. Стартираме командния интерпретатор **`cmd.exe`**:

```
Start | Programs | Accessories | Command Prompt
```

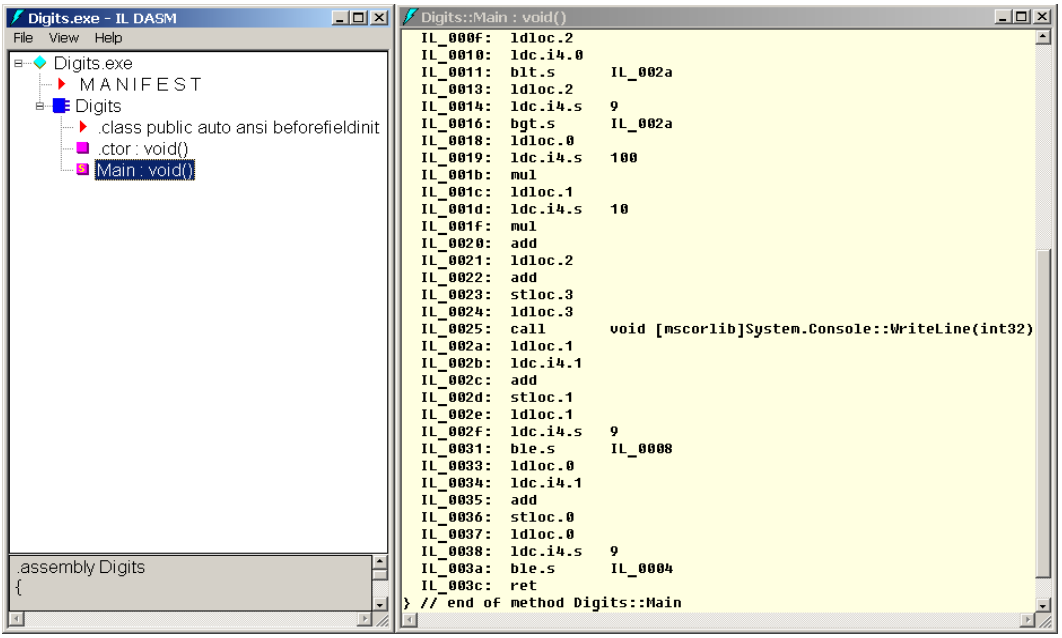
2. Отиваме в директорията, където се намира компилираната програма, например, програмата от предходния пример **Digits.exe**:

```
cd "C:\DotNet-course-lectures\Lecture-2-Introduction-to-CSharp\Demo-3-Digits\bin\Debug"
```

3. Извикваме от командната линия инструмента ILDASM (**ildasm.exe**) и му подаваме като параметър компилираната програма **Digits.exe**:

```
ildasm Digits.exe
```

4. Навигирайки по дървото, което ILDASM показва за асемблито **Digits.exe**, можем да видим как изглежда MSIL кодът за конструктора на класа **Digits** и за метода му **Main()**:



XML документация в C#

XML документацията в C# програмите представлява съвкупност от коментари, започващи с ///. Тя може да съдържа различни XML тагове – например, таг описващ връщана стойност на метод, таг за препратки към други методи и др. Като идея XML документацията прилича на JavaDoc в Java.

XML документацията значително улеснява поддръжката – документацията е част от кода, а не стои във външен файл. Поддържа се лесно и ползата от нея е видима още при разработката на приложението – Visual Studio .NET показва краткото описание на даден метод, ако той е документиран чрез вградената XML документация, при задействане на IntelliSense. C#

компилаторът може да извлича XML документацията като XML файл за по-нататъшна обработка.

Ето пример за използване на XML документация:

```
/// <summary>
/// The main entry point for the application.
/// </summary>
/// <param name="args">The command line arguments</param>
static void Main(string[] args)
{
   // ...
}

/// <summary>Calculates the square of a number</summary>
/// <param name="num">The number to calculate</param>
/// <returns>The calculated square</returns>
/// <exception cref="OverflowException">Thrown when the
/// result is too big to be stored in an int</exception>
/// <seealso cref="System.Int32" />
public static int square(int num)
{
   // ...
}
```

Тагове в XML документацията

Ето по-важните тагове, използвани в XML документацията в C#:

- **<summary>…</summary>** – кратко описание за какво се отнася даден тип, метод, свойство и т.н. Visual Studio .NET показва това описание при задействане на IntelliSense.
- **<remarks>…</remarks>** – подробно описание на даден тип, метод, свойство и т.н. Visual Studio .NET показва това описание в областта Object Browser.
- **<param name="…">…</param>** – описание на един от параметрите на даден метод.
- **<returns>…</returns>** – описание на връщаната от даден метод стойност.
- **<exception cref="…">…</exception>** – описание на изключение, което може да възникне в даден метод.
- **<seealso cref="…"/>** – препратка към информация, свързана с текущото описание.
- **<value>…</value>** – описание на свойство (property).

Извличане на XML документация от C# сорс код

Сега ще покажем как чрез C# компилатора може да се извлече документацията от C# файл в отделен XML файл. Нека имаме следната програма на C#, която използва XML документация:

MainClass.cs

```
using System;

namespace XMLCommentsDemo
{
   /// <summary>
   /// MainClass is a sample illustrating how to use XML
   /// documentation in C#.
   /// </summary>
   class MainClass
   {
      /// <summary>Calculates the square of a number</summary>
      /// <param name="num">The number to calculate</param>
      /// <returns>The calculated square</returns>
      /// <exception cref="OverflowException">Thrown when the
      /// result is too big to be stored in an int</exception>
      /// <seealso cref="System.Int32" />
      public static int Square(int num)
      {
         checked
         {
            return num*num;
         }
      }

      /// <summary>
      /// The main entry point for the application.
      /// </summary>
      /// <param name="args">The command line arguments</param>
      static void Main(string[] args)
      {
         Console.WriteLine("3*3 = " + Square(3));
      }
   }
}
```

За да извлечем документацията от тази програма, трябва да изпълним следните стъпки:

1. Стартираме командния интерпретатор **cmd.exe**:

   Start | Programs | Accessories | Command Prompt

2. Отиваме в директорията, където се намира сорс кода на програмата. Нека тя е **Demo-6-XML-Comments**:

```
cd "C:\DotNet-course-lectures\Lecture-2-Introduction-to-CSharp\Demo-6-XML-Comments"
```

3. Извикваме компилатора на C#, за да компилира файла **MainClass.cs**, като му задаваме опцията за извличане на XML документацията в отделен файл:

```
csc MainClass.cs /doc:MainClassComments.xml
```

4. Отваряме получения **.xml** файл с Internet Explorer, за да разгледаме съдържанието му.

Ето как полученият XML файл:

MainClassComments.xml

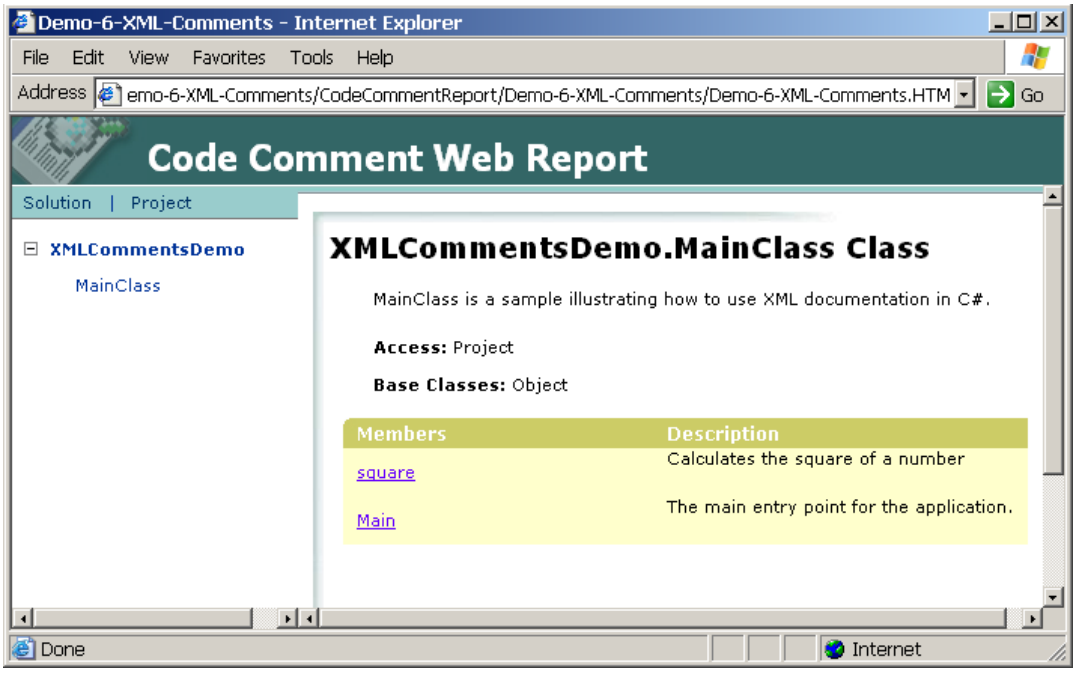
```
<?xml version="1.0"?>
<doc>
   <assembly>
      <name>MainClass</name>
   </assembly>
   <members>
      <member name="T:XMLCommentsDemo.MainClass">
         <summary>
         MainClass is a sample illustrating how to use XML
         documentation in C#.
         </summary>
      </member>
      <member name="M:XMLCommentsDemo.MainClass.Square(
            System.Int32)">
         <summary>Calculates the square of a number</summary>
         <param name="num">The number to calculate</param>
         <returns>The calculated square</returns>
         <exception cref="T:System.OverflowException">Thrown when
         the result is too big to be stored in an int</exception>
         <seealso cref="T:System.Int32"/>
      </member>
      <member name="M:XMLCommentsDemo.MainClass.Main(
            System.String[])">
         <summary>
         The main entry point for the application.
         </summary>
         <param name="args">The command line arguments</param>
      </member>
   </members>
</doc>
```

Генериране на HTML документация от VS.NET

Сега ще покажем как чрез Visual Studio .NET може да се генерира HTML документация за даден проект на C# по XML коментарите в неговия сорс код. Във вид на HTML документацията е много по-удобна за четене и разглеждане.

За целта трябва да изпълним следните стъпки:

1. Отваряме с Visual Studio .NET проект, в който сме използвали XML документиране, например проекта **`Demo-6-XML-Comments.sln`**, който съдържа кода от предходния пример.
2. От меню **`Tools`** избираме **`Build Comment Web Pages`**…. Указваме директория, където да се генерира HTML документацията, и натискаме бутона [OK]. Visual Studio .NET ще генерира в посочената директория съвкупност от HTML файлове, които документират нашия проект и съдържат XML коментарите от сорс кода му, подредени в подходящ за разглеждане вид.
3. Разглеждаме HTML документацията, която Visual Studio .NET е генерирал. Можем да навигираме по пространствата от имена, типовете от проекта и отделните му методи:



Директиви на предпроцесора

Нека сега разгледаме някои по-важни директиви на т. нар. предпроцесор. Преди компилация C# програмите преминават през процес на обработка, който идентифицира кода, който трябва да бъде компилиран при условна

компилация. Този процес се изпълнява от предпроцесора. Програмно върху предпроцесора можем да указваме влияние чрез т.нар. директиви – запазени думи, започващи със символа **#**.

Директиви за форматиране на сорс кода

В C# са въведени директиви за форматиране на сорс кода – **#region** и **#endregion**, които ограждат блок от кода, който се "свива" от редактора на Visual Studio .NET:

```
#region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
   // ...
}
#endregion
```

Visual Studio .NET редакторът много често слага региони, за да отдели автоматично-генерирания код от сорс кода, писан от програмиста. Директивите **#region** и **#endregion** се игнорират от C# компилатора и се използват единствено от средите за разработка.

Директиви за условна компилация

Директивите **#define** и **#ifdef** служат за условна компилация. Чрез тях може да се укаже на компилатора да компилира кода по различен начин според процесора, платформата и въобще средата, в която се извършва компилацията. Чрез **#if**, **#else**, **#elif**, **#endif** се задават границите на блоковете за условна компилация и съответните условия (знаци) за компилиране. Директивите **#define** и **#undef** дефинират знаци за условна компилация, според които се определя кой от блоковете за условна компилация да се разглежда.

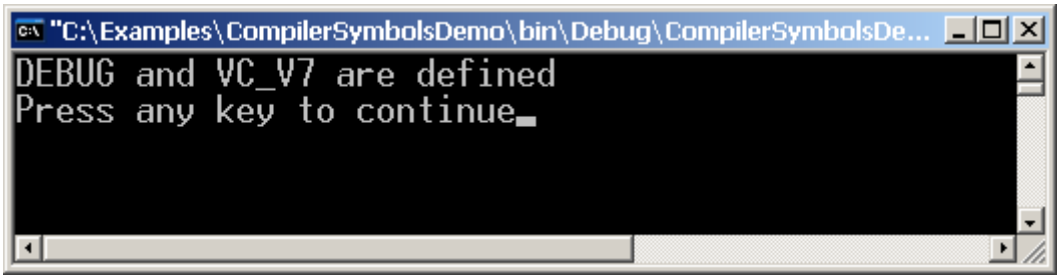
Условна компилация – пример

Следният пример показва как могат да се използват директивите на предпроцесора за условна компилация:

```
#define DEBUG
#define VC_V7

using System;
public class MyClass
{
   public static void Main()
```

```
   {
#if (DEBUG && !VC_V7)
     Console.WriteLine("Only DEBUG is defined");
#elif (!DEBUG && VC_V7)
     Console.WriteLine("Only VC_V7 is defined");
#elif (DEBUG && VC_V7)
     Console.WriteLine("DEBUG and VC_V7 are defined");
#else
     Console.WriteLine("DEBUG and VC_V7 are not defined");
#endif
   }
}
```

Ето и резултата от изпълнението на примера:



Директиви за контрол над компилатора

Директивите **#warning** и **#error** предизвикват предупреждения и грешки по време на компилация. Например следната програма на C# се компилира успешно, но с предупреждение:

```
#define DEBUG
public class MyClass
{
   public static void Main()
   {
#if DEBUG
#warning DEBUG symbol is defined
#endif
   }
}
```

Документацията на .NET Framework

Програмирането с .NET Framework е немислимо без неговата документация. Затова нека сега разгледаме какво представлява тя и как можем да я използваме при търсене на помощна информация по време на разработката на .NET приложения.

MSDN Library

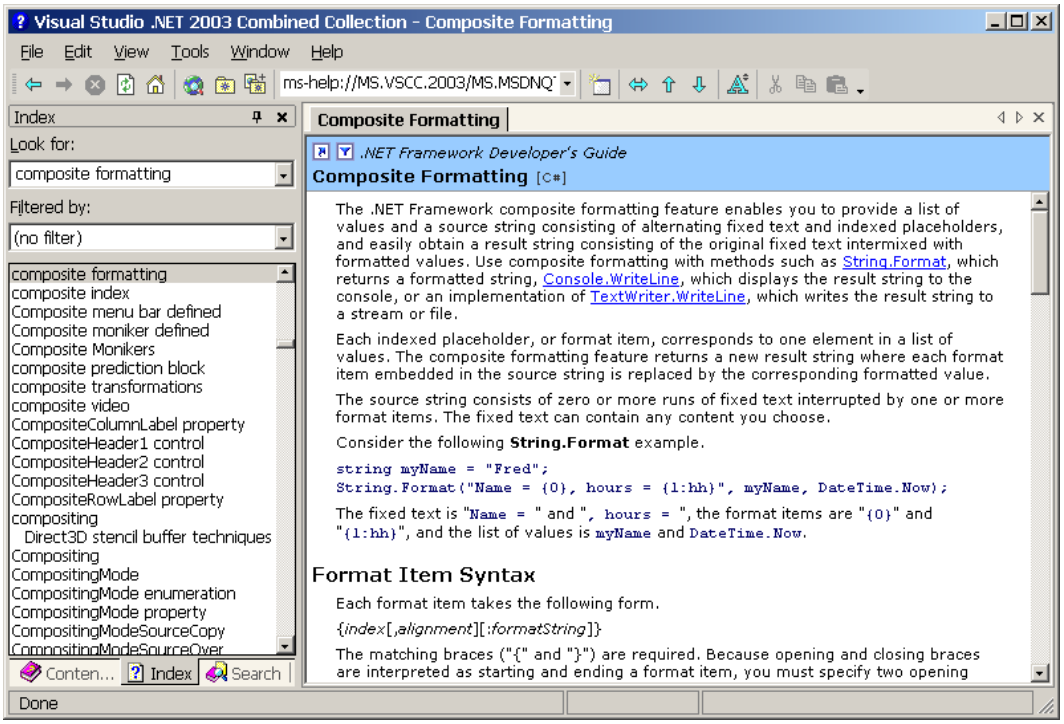
Документацията на .NET Framework се съдържа в "Microsoft MSDN Library".

MSDN Library е система, която предоставя пълен набор от технически документи, описващи продуктите, инструментите и технологиите за разработка на Microsoft (в частност .NET Framework и C#), както и средства за навигация и търсене в тях. MSDN Library съдържа технически ръководства, справочна информация, статии, примери и други ресурси за софтуерни разработчици.

MSDN Library е достъпен безплатно в on-line вариант от Интернет сайта за разработчици на Microsoft – http://msdn.microsoft.com/library/. Продуктът се разпространява и за локална инсталация заедно с партньорските програми на Microsoft.

MSDN Library – пример

За пример ще покажем как можем да намерим подробна информация за форматиращите низове в .NET Framework и тяхното използване. За целта стартираме MSDN Library и търсим "composite formatting":

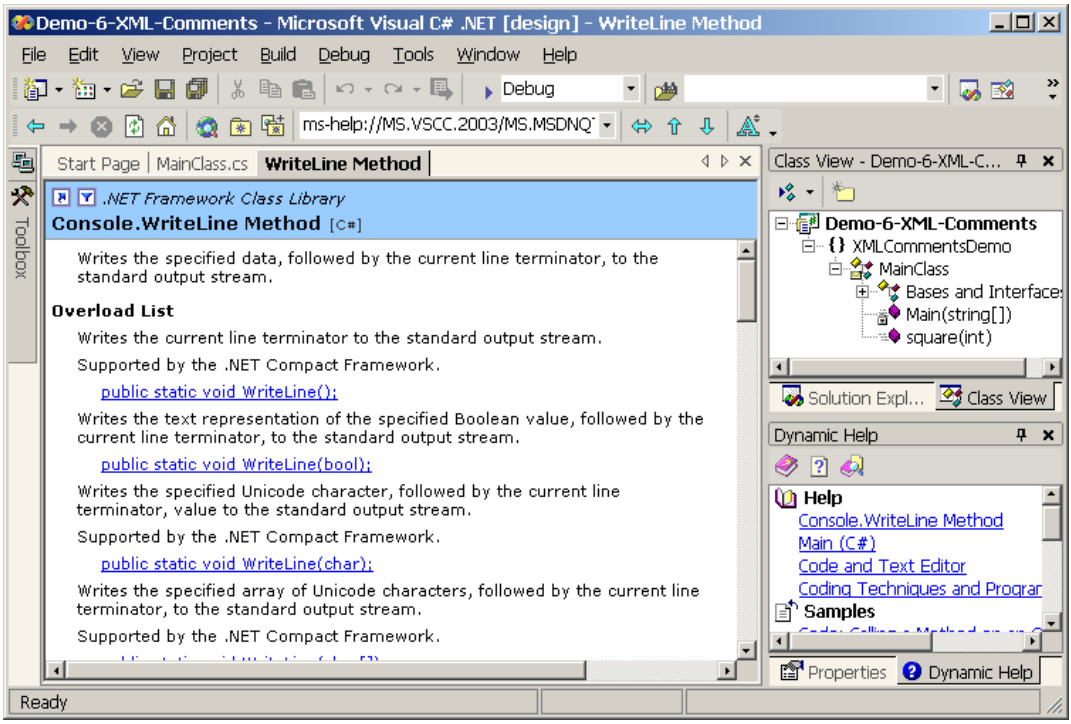


.NET Framework и MSDN Library

Документацията на .NET Framework е част от MSDN Library и се разпространява заедно с VS.NET и .NET Framework SDK.

Когато бъде инсталирана, документацията за .NET Framework, тя се интегрира във VS.NET и може да се използва директно от него. Например, ако се нуждаем от помощна информация за метода **`WriteLine(…)`** на класа **`Console`**, натискаме [F1] във Visual Studio .NET докато курсорът е върху този метод. Отваря се нов прозорец, в който са описани параметрите, типа на връщаната стойност, типовете изключения, които може да предизвика описвания метод, в кое пространство от имена се намира и др.

Ето как изглежда описанието на метода **`WriteLine(…)`** на класа **`Console`**:



Упражнения

1. Съставете програма на C#, която въвежда от конзолата име на студент и го поздравява в стил "Здравей, <име>!".
2. Съставете програма на C#, която въвежда коефициентите на квадратно уравнение и пресмята реалните му корени.
3. Напишете програма, която намира всички символни низове, които се състоят от точно 5 малки латински букви и са симетрични спрямо средата си.
4. Проследете работата на програмата от задача 3 с дебъгера на Visual Studio .NET.

5. Променете програмата от задача 3, така че да намира само тези низове, които съдържат четен брой гласни букви. Колко са тези низове?
6. Добавете XML документация в програмата от задача 5 и генерирайте HTML документация от Visual Studio .NET.
7. Напишете програма, която намира сумарната стойност на група фактури. Програмата трябва да въвежда последователно от конзолата сумите на фактурите (реални числа със знак) докато стигне до празен ред. Сумарната стойност на фактурите трябва да се отпечата в 10-символно поле, дясно подравнена, с точност 2 знака след десетичната запетая (потърсете в документацията подходящ форматиращ стринг).
8. Напишете програма, която прочита прост числен израз, състоящ се от реални числа, свързани с операциите "+" и "-", и изчислява и отпечатва стойността му.

Използвана литература

1. Светлин Наков, Въведение в C# – http://www.nakov.com/dotnet/lectures/Lecture-2-Introduction-to-CSharp-v1.0.ppt
2. MSDN Training, Programming C# (MOC 2124C), Module 2: Overview of C#
3. MSDN Training, Programming C# (MOC 2124C), Module 3: Using Value-Type Variables
4. Jessy Liberty, Programming C#, Second Edition, O'Reilly, 2002, ISBN 0-596-00309-9
5. Svetlin Nakov, .NET Framework Overview – http://www.nakov.com/publications/Nakov-DotNET-Framework-Overview-english.ppt
6. MSDN, C# Keywords – http://msdn.microsoft.com/library/en-us/csref/html/vclrfcsharpkeywords_pg.asp
7. MSDN, C# Built-in Types Table – http://msdn.microsoft.com/library/en-us/csref/html/vclrfbuiltintypes.asp
8. MSDN, Common Type System Overview – http://msdn.microsoft.com/library/en-us/cpguide/html/cpconcommontypesystemoverview.asp
9. MSDN, Enumerations – http://msdn.microsoft.com/library/en-us/cpguide/html/cpconEnumerations.asp
10. MSDN, C# Operators – http://msdn.microsoft.com/library/en-us/csref/html/vclrfCSharpOperators.asp
11. MSDN, Statements (C# Programmer's Reference) – http://msdn.microsoft.com/library/en-us/csref/html/vclrfstatements.asp

12. MSDN, XML Documentation Tutorial (C# Programmer's Reference) – http://msdn.microsoft.com/library/en-us/csref/html/vcwlkxmldocumentationtutorial.asp
13. MSDN, C# Preprocessor Directives - http://msdn.microsoft.com/library/en-us/csref/html/vclrfPreprocessorDirectives.asp
14. MSDN, Composite Formatting – http://msdn.microsoft.com/library/en-us/cpguide/html/cpconcompositeformatting.asp
15. MSDN, Console Class (.NET Framework) – http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemconsoleclasstopic.asp

Глава 3. Обектно-ориентирано програмиране в .NET

Необходими знания

- Познаване на принципите на обектно-ориентираното програмиране
- Познаване на поне един обектно-ориентиран език за програмиране – C++, Java, C#, Object Pascal/Delphi

Съдържание

- Предимства и особености на ООП.
- Основни принципи на ООП. Основни понятия
- ООП и .NET Framework
- Членове на клас
- Член-променливи (полета). Константни полета
- Методи (член-функции)
- Статични членове
- Конструктори. Статичен конструктор
- Предаване на параметрите
- Свойства. Индексатори
- Предефиниране на оператори
- Наследяване
- Интерфейси. Абстрактни класове
- Виртуални членове. Предефиниране и скриване.
- Клас диаграми
- Принципи при обектно-ориентирания дизайн
- Пространства от имена (namespaces)

В тази тема ...

В настоящата тема ще направим кратък обзор на основните принципи на обектно-ориентираното програмиране (ООП) и средствата за използването им в .NET Framework и езика C#. Ще се запознаем с типовете "клас", "структура" и "интерфейс" в C#. Ще въведем понятието "член на тип" и

ще се разгледаме видовете членове (член-променливи, методи, конструктори, свойства, индексатори и др.) и тяхната употреба. Ще се спрем и на наследяването на типове в различните му аспекти и приложения. Ще обърнем внимание и на полиморфизмът в C# и свързаните с него понятия и програмни техники. Накрая ще обсъдим някои утвърдени практики при създаването на ефективни йерархии от типове.

Предимства и особености на ООП

Обектно-ориентираното програмиране се е наложило като стандарт при почти всички съвременни езици за програмиране. То предоставя мощно средство за моделиране на обектите от реалния свят и взаимоотношенията между тях, позволява добро структуриране на програмния код и улеснява неговото преизползване. Благодарение на капсулацията на данните, чрез която се скриват имплементационните детайли и се намалява сложността на софтуера, както и на възможностите за наследяване на свойства и действия и за работа с абстрактни данни и изпълнение на абстрактни операции, ООП е се е утвърдило като предпочитан подход при създаване на големи приложения и библиотеки.

Моделиране на обекти от реалния свят

В основата на ООП стоят обектите, моделиращи обекти от реалния свят и взаимодействията между тях. Това позволява изграждането на софтуерни системи, които въпреки сложността си са разбираеми и в следствие на това - лесни за разширяване и поддръжка. Даден обект, представящ същност (entity) от реалния свят, би могъл (почти) без изменения да играе ролята на същия физически обект в друга софтуерна система.

Обектите притежават атрибути, които описват свойствата на им, и операции – възможните действия, които могат да се извършват с обекта.

Преизползване на програмния код

Едно от основните предимства на обектно-ориентирания подход е, че позволява лесно **преизползване на програмния код** (**code reuse**). Това се постига с помощта на **наследяване** и **полиморфизъм**, които ни позволяват да дефинираме общите свойства и действия за множество от типове обекти само в един от тях.

Основни принципи на ООП

Трите основни принципа на ООП са капсулация на данните, наследяване и полиморфизъм. Те са основните характеристики, които определят един език за програмиране като обектно-ориентиран.

Капсулация на данните

Основна концепция в ООП е обектът да се разглежда като "черна кутия" – използващите обекта "виждат" само атрибутите и операциите, които са присъщи на обекта от реалния свят, без да се интересуват от конкретната им реализация – клиентът на обекта трябва да знае само какво може да прави обектът, а не как го прави. В такъв смисъл "капсулация" означава скриване на ненужните детайли за обектите и откриване към външния свят само на важните техни характеристики и свойства.

Обектите в ООП съдържат своите данни и средствата за тяхната обработката, капсулирани като едно цяло.

Наследяване

Ако един обект съдържа всички свойства и действия на друг, първият може да го наследи. По този начин наследеният обект освен собствените си атрибути и операции приема и тези на "родителя" си (базовия клас), като така се избягва повторното им дефиниране и се позволява създаването на йерархии от класове, моделиращи по естествен начин зависимостите от реалността.

За да изясним това понятие, ще си послужим с класическият в ООП пример за класа от обекти **Animal**, който представлява абстракция за множеството от всички животни. Всички обекти от този клас имат общи характеристики (например цвят и възраст) и обща функционалност, например операциите **Eat** и **Sleep**, докато за класът **Dog**, представляващ множеството от всички кучета, които също са животни, би могъл да предоставя операциите **Eat**, **Sleep** и **Bark**. Удачно е класът **Dog** да наследи **Animal** – тогава той ще съдържа описание само на собственото си действие **Bark**, докато тези на **Eat** и **Sleep** ще получи от базовия си клас.

Чрез наследяването се постига **специализация**, или конкретизация на класовете, тъй като базовият клас представлява категория от обекти по-обща от тази на наследяващите го. Ако си послужим с горния пример, множествата на кучетата и котките са подмножества на множеството от всички животни.

Може да се каже, че наследяването моделира "is-a" отношението между обектите, например можем да твърдим, че кучето е животно, тъй като то "може да прави" всичко, което и животното и притежава всички животински характеристики (цвят, възраст и т.н.).

Полиморфизъм

Полиморфизъм буквално означава приемането на различни форми от един обект. Нека е даден базов клас, представящ категория от обекти, които реализират общо действие, което се наследява от множество класове, описващи по-тесни категории. Въпреки, че те всички споделят това действие, те могат да го реализират по различен начин. Когато разполагаме с обекти от базовия клас, знаем че всички те реализират това действие, независимо на кой наследен клас принадлежат. Поради това можем да го използваме без да се интересуваме от конкретната му реализация. Например, ако класът **Animal** предоставя действието **Talk** и разгледаме наследяващите го класове **Dog** и **Cat**, всеки от тях го реализира по конкретен начин. И ако имаме животно, което издава звук и то е куче – ще лае, а ако е котка – ще мяучи.

Полиморфизмът позволява унифицираното извършване на действие над различни обекти, които го реализират. В този случай издаването на звук

от животно е **полиморфно действие** – такова, което се реализира по различен начин в различните наследници на базовия клас.

Основни понятия в ООП

Без да претендираме за изчерпателност ще даден кратка дефиниция за основните понятия от ООП, които ще използваме по-нататък. Ако откривате, че повечето от тези термини са ви напълно непознати, ви препоръчваме първо да се запознаете с принципите на обектно-ориентираното програмиране от някоя специализирана книга по ООП, а след това да продължите нататък. В настоящата тема ще направим преглед на реализацията на ООП в .NET Framework, а не на ООП като идеология.

Клас

Класовете са категории от обекти, споделящи общи свойства и операции, които могат да се извършват върху тях. Например класът "студент" представя множеството от всички студенти. Класът не съществува реално като физическа същност, а по-скоро можем да го разгледаме като описание на неговите обекти.

Обект

Обект наричаме конкретен елемент от даден клас (инстанция), например студентът Тодор Георгиев, трети курс, ядрена физика в СУ.

Инстанциране

Процесът на създаване на обект от даден клас е инстанциране. Обектите, създадени при инстанциране на даден клас, се наричат негови **инстанции**. Например в резултат от инстанцирането на класа "студент" можем да получим обекта "Иван Петров", който е инстанция на класа "студент".

Свойство

Свойство се нарича видима за външния свят характеристика (атрибут) на обектите от даден клас. Например свойства на класа "студент" са личните имената му, личните му данни, оценките му и др.

Метод

Метод е действие, което всички обекти от даден клас могат да извършват. Например всички обекти от класа "студент" могат да извършват действието "явяване на изпит".

Интерфейс

Интерфейсът е описание на съвкупност от действия, които даден обект може да извършва. Ако един обект може да извършва всички действия от

даден интерфейс, казваме че обектът **реализира**, или **имплементира** интерфейса. Класът "студент", например, би могъл да реализира интерфейса "учащ" съдържащ действието "учене".

Наследяване на класове

Наследяване в ООП наричаме възможността един клас, наричан **наследник**, да придобие свойства и действия на друг клас – **родител** (**базов** клас). Например класът "прекъснал студент" би могъл да наследи класа "студент", като към наследените методи и свойства добави собствени, например "получаване на призовка от военните власти".

Абстракция на данните

Абстракция на данните наричаме възможността да работим с данни без да се интересуваме от тяхното вътрешно представяне, а само от операциите, които можем да извършваме над тях. Удачно е този подход да се осъществи чрез използването на интерфейси.

Структури от данни, които дефинират група от операции, но не разкриват информация как са имплементирани тези операции, се наричат **абстрактни структури от данни**.

Абстракция на действията

Абстракцията на действията е възможността да изпълняваме действия, без да се интересуваме от конкретната им реализация. Обикновено се постига чрез полиморфизъм. Например ако извикваме даден метод от даден клас през неговия базов клас или интерфейс, ние реално извикваме абстрактно действие от базовия клас, което е реализирано в класа-наследник.

ООП и .NET Framework

В .NET Framework обектно-ориентираният подход е залегнал на най-дълбоко архитектурно ниво. Всеки тип, дефиниран от потребителя, и всички типове от Common Type System (CTS) наследяват `System.Object` или негов наследник.

В някои обектно-ориентирани езици се използват примитивни типове данни (булеви, числови, символни), които в езиците от .NET Framework са също наследници на `System.Object`.

Всички .NET езици са обектно-ориентирани и приложенията се пишат изцяло обектно-ориентирано – няма глобални функции и всички действия се извършват или чрез създаване на обекти и с използване на методите и свойствата им, или чрез използване на **статични** членове (тях ще разгледаме малко по-нататък).

Типове данни

В предходната глава въведохме понятието **тип** и разделихме типовете в C# на типове стойностни и референтни. Следва да представим една по-подробна класификация на типовете данни в .NET Framework. Те биват:

- класове
- структури
- интерфейси
- делегати
- изброени типове

Реализация на понятието клас

Понятието "клас" от ООП се реализира в .NET Framework чрез класове (classes) и структури (structs).

Не трябва да бъркаме понятието клас от концепциите на ООП с понятието клас в .NET Framework. Разликата е тънка – **класът** в .NET действително е клас според ООП терминологията, но обратното не е вярно. ООП терминът клас се реализира и по още един начин – чрез **структури**.

Основната разлика между класовете и структурите в .NET Framework е, че структурите са стойностни типове, докато класовете са референтни типове. Структурите по-интуитивно моделират данни, от които се очаква поведение като на примитивни типове, докато класовете по-добре моделират обекти от реалния свят, които могат да извършват определени действия.

Тъй като типовете по стойност в общия случай се създават в стека за изпълнение на програмата, структурите е добре да съдържат малки по-обем данни, а по-големите количества е удачно да се обработват с помощта на класове, инстанциите на които съхраняват членовете си в динамичната памет.

Множествено наследяване

Някои обектно-ориентирани езици позволяват използването на множествено наследяване – възможността един клас да приеме методи и свойства от няколко родителя. При проектирането на .NET Framework е взето решение това да не се допуска.

Една от причините в .NET Framework да няма множествено наследяване е, че множественото наследяване води до конфликти, например ако един клас наследи елемент с едно и също име от повече от един родител.

От друга страна множественото наследяване води до по-сложни и трудно разбираеми йерархии – такива, образуващи граф, докато при наследяването от единствен родител се получава дърво.

В .NET Framework приемането на характеристики и поведение от повече от една същности от реалния свят се осъществява чрез реализиране на няколко интерфейса едновременно, при което обаче не може да се наследят данни или програмен код, а само дефиниции на действия.

Класове

Класовете в C# са основните единици, от които се състоят програмите. Те моделират обектите от реалния свят и могат да дефинират различни членове (член-променливи, методи, свойства и др.). Нека видим как изглежда един примерен клас на езика C#:

```
class Student
{
   // Private member declarations
   private string mFirstName;
   private string mLastName;
   private string mStudentId;

   // Constant
   private const double PI = 3.14159265358979323846264433;

   // Constructor
   public Student(string aStudentId)
   {
        mStudentId = aStudentId;
   }

   // Property
   public string FirstName
   {
      get
      {
         return mFirstName;
      }
      set
      {
         mFirstName = value;
      }
   }

   // Read-only property
   public string StudentId
   {
      get
      {
         return mStudentId;
      }
   }
```

```
   // Method
   public string StoreExamResult(
      string aSubject, double aGrade)
   {
      // ...
   }
}
```

В горния пример е дефиниран класът **Student**, илюстриращ някои от видовете членове, които класовете могат да реализират – капсулираните полета **mFirstName**, **mLastName** и **mStudentId**, константата **PI**, конструкторът **Student(…)**, свойствата **FirstName** и **StudentId** и методът **StoreExamResult(…)**. С течение на темата ще се запознаем по-отблизо с всеки от тези видове членове.

Членове на тип

В .NET типовете "клас" и "структура", като реализация на понятието клас от ООП, могат да съдържат в себе си членове (members), подобно на други обектно-ориентирани езици като Java и C++. Членовете могат да бъдат от един от следните видове:

- полета, или член-променливи (fields)
- константи (constants)
- методи, или член-функции (methods)
- свойства (properties)
- индексатори (indexers)
- събития (events)
- оператори (operators)
- конструктори (constructors)
- деструктори (destructors)
- вложени типове (класове, структури, изброени типове и др.)

Видимост на членовете

Множеството от типове, които могат да "виждат" определен член на даден клас се определя от видимостта. Правилното задаване на видимостта на членовете е ключов момент в разработването на йерархии от класове, тъй като основен принцип в ООП е клиентът на класа да вижда само това, което му е необходимо, и нищо повече. Следва описание на нивата на видимост в .NET Framework.

public

Глобална видимост – членовете с такова ниво на достъп могат да се достъпват от всеки тип.

protected internal

Това са членовете, видими от всички типове, дефинирани в асемблито, в което е дефиниран дадения, а също и от наследниците на типа.

internal

Членове, които се достъпват от всички типове, дефинирани в асемблито, в което е дефиниран дадения.

protected

Членове, видими само от наследниците на дадения тип.

private

Капсулирани членове, видими единствено в рамките на типа.

Член-променливи

Данните, с които инстанцията на класа работи, се съхраняват в **член-променливи** (или още **полета)**. Те се дефинират в тялото на класа и могат да се достъпват от други видове членове – методи, конструктори, индексатори, свойства. В следващия пример ще покажем няколко декларации на член-променливи, за които в последствие ще дадем обяснения.

```
class Student
{
   private string mFirstName;
   private string mLastName;
   private string mStudentId;
   private int mCourse = 1;
   private string mSpeciality;
   private Course[] mCoursesTaken;

   // Avoid missing the visibility modifier
   string mRemarks = "(няма забележки)";
}
```

Дефиниране на ниво на видимост

Дефиницията на всяко поле започва с ниво на видимост. Допустими са всички по-горе изброени нива на видимост, но в примера са използвани само **`private`**, защото скриването на полетата от използващите класа, т.е. указването на видимост **`private`** или **`protected`**, е утвърдена практика в ООП. Когато искаме да предоставим данните на класа на околния свят в

.NET е прието вместо полета с ниво на достъп "**public**" да се използват свойства, на които ще се спрем малко по-късно. Степента на видимост може и да не бъде определена явно, както е в последния ред за полето **mRemarks** от примера и в този случай се подразбира **private**. Тази практика не се препоръчва, защото води до по-неясен код.

Дефиниране на тип

Следващият елемент от дефиницията на член-променлива е типът, който се указва задължително. Може да бъде произволен .NET тип от CTS или дефиниран от потребителя.

Задаване на име

След типа следва името на дефинираното поле, чрез което се обръщаме към него. То представлява идентификатор, т. е. последователност от unicode символи – главни и малки букви, цифри, -(тире) и _(подчертаващо тире), незапочваща с цифра или тире.

Имената на полетата и въобще на членовете в .NET Framework могат да бъдат идентични със съществуващи имена на типове или пространства от имена (на тях ще се спрем в края на темата). Например класът **Student** може да има свойство със същото име **Student**. Могат да бъдат и запазени думи, но само ако бъдат предшествани от **@**. Допуска се и използването на нелатински букви в имената, но не се препоръчва.

Задаване на стойност

При дефиницията на поле можем да му зададем стойност, както в примера това е направено за **mCourse** и **mRemarks**. Ако началната стойност бъде пропусната, на член-променливата се задава стойност по подразбиране. За референтните типове това е **null**, а за стойностните типовете е **0** или неин еквивалент (например **false** за **boolean**). В .NET Framework всички членове и променливи се инициализират автоматично. Това намалява грешките, възникващи заради използването на неинициализирани променливи.

Константни полета

Константните полета (или само **константи**) много приличат на обикновените полета, но имат някои особености. Нека обърнем внимание на следния пример, който показва няколко дефиниции на константи:

```
public class MathConstants
{
   public const string PI_SYMBOL = "π";
   public const double PI = 3.1415926535897932385;
   public const double SQRT2 = 1.4142135623731;
}
```

От примера виждаме, че дефиницията на константа е дефиницията на поле с добавена ключовата дума **const**. Има и някои други разлики.

При декларирането на константно поле е задължително да се предостави стойност. Освен това стойността на константата не може да бъде променяна по време на работата с типа, в който е дефинирана – може само да бъде прочетена. Константите реално не съществуват като полета в типа, а съществуват само в сорс кода и се заместват със стойността им по време на компилация. Поради тази причина **const** декларациите в C# се наричат още **compile-time константи**, т. е. константи, които съществуват само по време на компилацията.

Полета само за четене

Друг специален вид полета, подобни на константите, са полетата само за четене (read-only fields). Те се различават от константните по това, че стойността им освен при дефиницията може да бъде зададена и в конструктор, но от там нататък не може да бъде променяна. Член-променлива само за четене се декларира, като се използва запазената дума **readonly**, като в примера:

```
class ReadOnlyDemo
{
   private readonly int mSize;

   public ReadOnlyDemo(int aSize)
   {
      mSize = aSize; // cannot be further modified!
   }
}
```

За разлика от константите, полетата само за четене са реални полета в типа, които обаче, задължително трябва да се инициализират в конструктора на класа или при деклариране, защото след това не може да им бъде присвоявана стойност и биха останали с подразбиращата се. Поради тази причина те се наричат още **run-time константи**, т. е. константи, които се инициализират по време на изпълнение на програмата.

Методи

Методите (или още **член-функции**) дефинират операции за типа, в който са дефинирани. Те могат да боравят с членовете му, независимо от степента им на видимост, да ги достъпват и променят (освен полетата обявени като константни или само за четене).

В C# функции могат да бъдат дефинирани единствено като членове на клас или структура, за разлика от други обектно-ориентирани езици, където се използват **глобални функции** – такива, които не са обвързани с конкретен тип и са общодостъпни. В C# функции, които се достъпват без

да е нужна инстанция на даден клас, се дефинират като **статични**. На тях ще се спрем след малко.

Задаване на видимост

Подобно на полетата, и методите могат да имат ниво на видимост. И синтактично, и от гледна точка на стила на програмиране, на методите е допустимо да се зададе коя да е от възможните нива на видимост, тъй като те представляват действията с типа и за някои от тях е необходимо да бъдат видими за околния свят, а за други – не. Отново подразбиращото се ниво на видимост е **`private`**, но е препоръчително да се декларира изрично.

Параметри и върната стойност

Методите могат да приемат параметри и да връщат стойност. Параметрите имат тип, който може да бъде всеки валиден .NET тип. Върнатата стойност може да бъде също от всеки възможен тип, а може и да отсъства. Нека обърнем внимание на следния пример:

```
class MethodsDemo
{
   public void SayHiGeorgi()
   {
      SayHi("Гошо");
   }

   public void SayHiPeter()
   {
      SayHi("Пешо");
   }

   private void SayHi(string aName)
   {
      if (aName == null || aName == "" )
      {
          return;
      }
      Console.WriteLine("Здравей, {1}", aName);
   }

   public int Multiply(int x, int y)
   {
      return x * y;
   }
}
```

Първите два метода, **`SayHiGeorgi()`** и **`SayHiPeter()`**, не приемат никакви параметри и не връщат стойност. Третият, **`SayHi(string aName)`**, приема един параметър от тип **`string`** и не връща стойност. Последният,

`Multiply(int x, int y)`, приема два параметъра от тип **`int`** и връща стойност също от тип **`int`**.

В дефинициите на първите три метода от примера забелязваме ключовата дума **`void`** – тя се използва при методи, които не връщат стойност. За методи, които връщат стойност, вместо ключовата дума **`void`** се указва типа на връщаната стойност.

В последния метод забелязваме как се употребява ключовата дума **`return`** за връщане на стойност. Същата ключова дума използваме и за прекратяване на изпълнението на метод, който не връща стойност, както в метода **`SayHi(aName)`**.

Методи с еднакви имена

В C# е допустимо един тип да има два и повече метода с едно и също име, но с някои ограничения. Ще въведем понятие, свързано с използването на едно и също име за няколко метода. Комбинацията от името, броя и типа на параметрите на метод наричаме **сигнатура**. Ако два метода имат едно и също име, те задължително трябва да се различават по сигнатура. Следващият пример илюстрира дефинирането на три метода с еднакви имена:

```
int Sum(int a, int b)
{
   return a + b;
}

int Sum(int a, int b, int c)
{
   return a + b + c;
}

long Sum(long a, long b, long c) // avoid this
{
   return a + b + c;
}
```

Горните дефиниции са напълно валидни – първите два метода се различават по броя на параметрите си, а вторият и третият – по типа.

Трябва да сме особено внимателни с дефиниции като последните две и е препоръчително да се избягват, тъй като не е очевидно кой метод ще бъде извикан при обръщение като `int sumTest = sum(1,2,3)`. Компилаторът по никакъв начин не ни предупреждава за двусмислието. В горния пример ще бъде извикан първият метод – `sum(int a, int b, int c)`.

Статични членове

Както вече споменахме, в C# функции, които могат да се извикват без да е нужна инстанция на клас, се реализират като **статични** (или **общи**) методи. Това става, като в дефиницията им включим ключовата дума **`static`**. Статичните членове се споделят от всички инстанции и се използват за пресъздаване на свойства и действия, които са постоянни за всички обекти от дадения клас. Достъпът до статичните членове на типа се извърша директно, а не през инстанция, както в следващия пример:

```
class Bulgaria
{
   private static int mNumberOfCities = 267;

   public static int NumberOfCities
   {
      get
      {
         return mNumberOfCities;
      }
   }

   public static void AddCity(string aCityName)
   {
      mNumberOfCities++;
      // ...
   }

   // ...

   static void Main()
   {
      Console.WriteLine(
         "В България има {0} града.", Bulgaria.NumberOfCities);
   }
}
```

В примера видяхме дефинирането и използването на статични полета, методи и свойства. Използвахме статичните свойства без да инстанцираме класа **`Bulgaria`** никъде.

Важна особеност, която трябва да имаме предвид при използването на статични методи и свойства, е че те могат да използват само статични полета. Полетата, които са обвързани с инстанция могат да се достъпват само в нейния контекст, а статичните методи и свойства са независими от инстанцията.

Статичните полета на типа много приличат на глобалните променливи в по-старите езици за програмиране като C, C++ и Pascal. Както глобалните

променливи, статичните полета са достъпни от цялото приложение и имат само една инстанция.

От членовете на типа, освен полетата, свойствата и методите също и конструкторите, индексаторите и събитията могат да бъдат статични. Константите също са общи за всички инстанции на типа, но не могат да бъдат статични. Деструкторите също не могат да бъдат статични, докато операторите задължително са.

Конструктори

Конструкторите се използват при създаване на обекти и служат за **инициализация**, или начално установяване на състоянието на полетата на обекта. Механизмът на работа и синтаксисът за дефиниране на конструкторите в C# са подобни на други обектно-ориентирани езици, като Java и C++ с някои особености, на които ще обърнем внимание. Допуска се използването на повече от един конструктор, като конструкторите трябва да се различават по броя и/или типа на параметрите. Възможно е и да не се дефинира конструктор и в такъв случай компилаторът създава подразбиращ се – публичен, с празно тяло и без параметри.

Инициализиране на полетата

Съществуват три възможности за инициализацията на полетата на обекта – да бъдат инициализират в конструктор, при декларацията им или да нямат изрично зададена стойност.

Инициализациите, описани в тялото на конструктора се изпълняват по време на изпълнението този конструктор – при създаване на обект от съответния клас с ключовата дума **new** в C#.

Инициализациите, дефинирани при декларацията на полетата се изпълняват директно преди конструктора. Можем да приемем, че при компилацията инициализациите на полетата се добавят в началото на всеки конструктор. Всъщност C# компилаторът прави точно това скрито от програмиста – поставя код, който инициализира всички член-променливи на типа във всички негови конструктори.

Полетата, които нямат зададена начална стойност, получават стойност по подразбиране (нулева стойност). Това поведение се изисква от спецификацията на езика C# и не зависи от конкретната имплементация на компилатора.

Конструктори – пример

Със следващия пример ще разгледаме примерни дефиниции на конструктори на базов клас с един наследник:

```
class Student
{
   private string mName;
```

```
   private int mStudentId;
   private string mPosition = "Student";

   public Student(string aName, int aStudentId)
   {
      mName = aName;
      mStudentId = aStudentId;
   }

   public Student(string aName) : this(aName, -1)
   {
   }

   public static void Main()
   {
      Student s = new Student("Бай Киро", 12345);
   }
}

public class Kiro : Student
{
   public Kiro() : base("Бай Киро", 12345)
   {
   }

   // ...
}
```

Забелязваме употребата на ключовите думи **this** и **base** след дефиницията на конструкторите на класа. Те представляват съответно обръщения към друг конструктор на същия клас и към конструктор на базовия клас, като в скобите се изреждат параметрите, които се подават на извиквания конструктор. В примера е използване наследяване, на което ще с спрем в детайли след малко (класът **Kiro** наследява класа **Student**).

Изследване на MSIL кода за конструкторите в C#

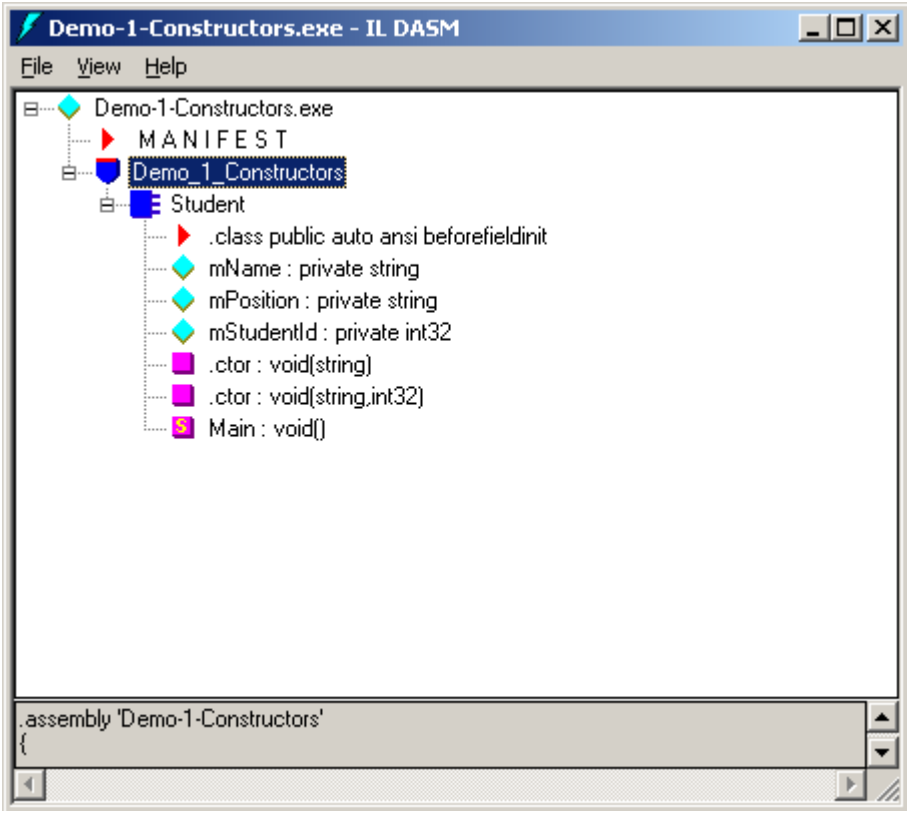
В следващата демонстрация ще си послужим с инструмента IL DASM (**ildasm.exe**), който е част от .NET Framework SDK, за да разгледаме MSIL кода, който C# компилаторът генерира за класа **Student**, който дефинирахме в примера по-горе. С това упражнение не само ще се запознаем с работата с инструмента, но и ще забележим особеностите в генерирания код, свързани с полетата със зададена стойност при декларацията. Ето стъпките, които трябва да направим:

1. Отваряме **Demo-1-Constructors.sln**, елементарен Visual Studio .NET проект с единствен C# файл, който съдържа кода от горния пример. Компилираме проекта.

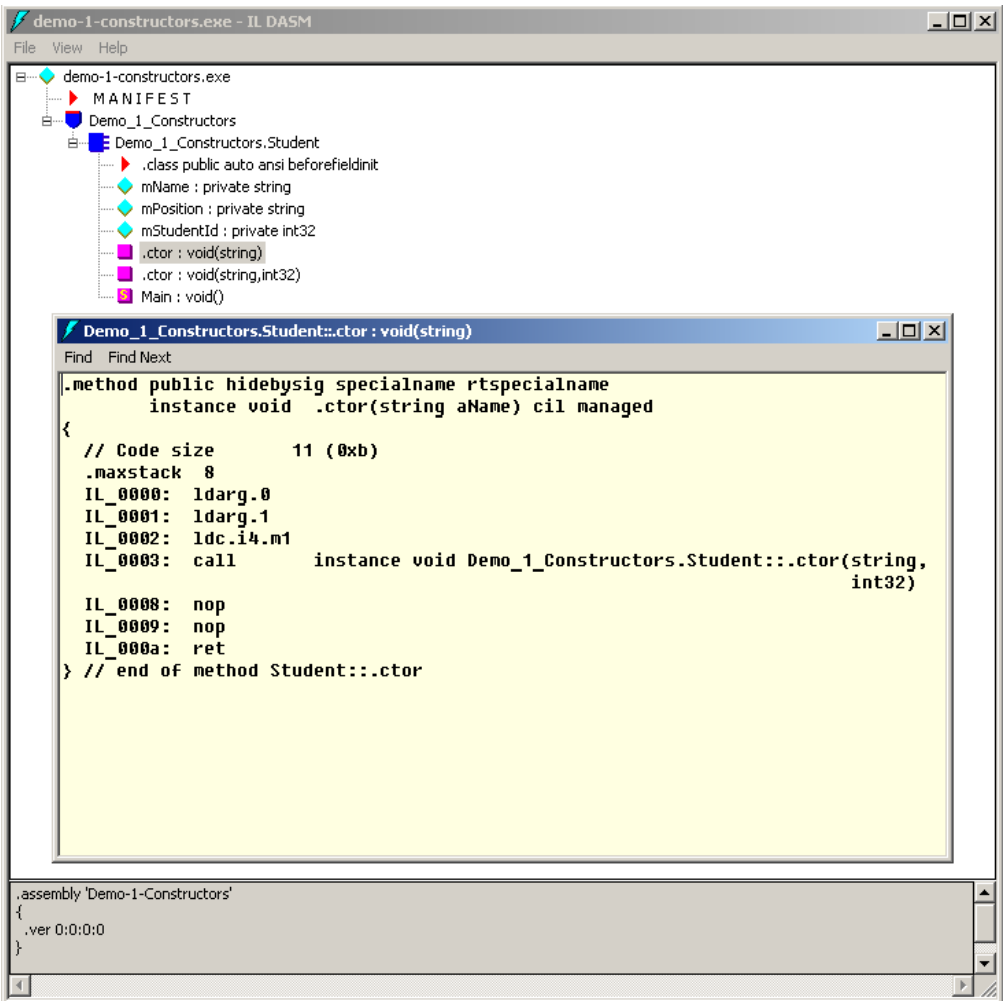
2. Стартираме командния интерпретатор към Visual Studio .NET. Не използваме стандартния **cmd.exe**, а този, който се намира в **Start -> Programs -> Microsoft Visual Studio 2003 -> Visual Studio Tools**, защото той се стартира с регистрирани пътища към .NET инструментите, които се използват от командния ред.
3. Избираме директорията, където се намира изпълнимият файл, получен при компилиране на проекта – **Demo-1-Constructors.exe**. Ако не сме променили настройките на Visual Studio .NET, това ще е директорията **<директория на проекта>\bin\Debug**.
4. Извикваме от командния ред инструмента **ildasm** и му подаваме като параметър компилираното приложение:

```
ildasm Demo-1-Constructors.exe
```

Ето как изглежда прозорецът на инструмента, в който е заредено асемблито от приложението, когато разпънем всички елементи от дървото:

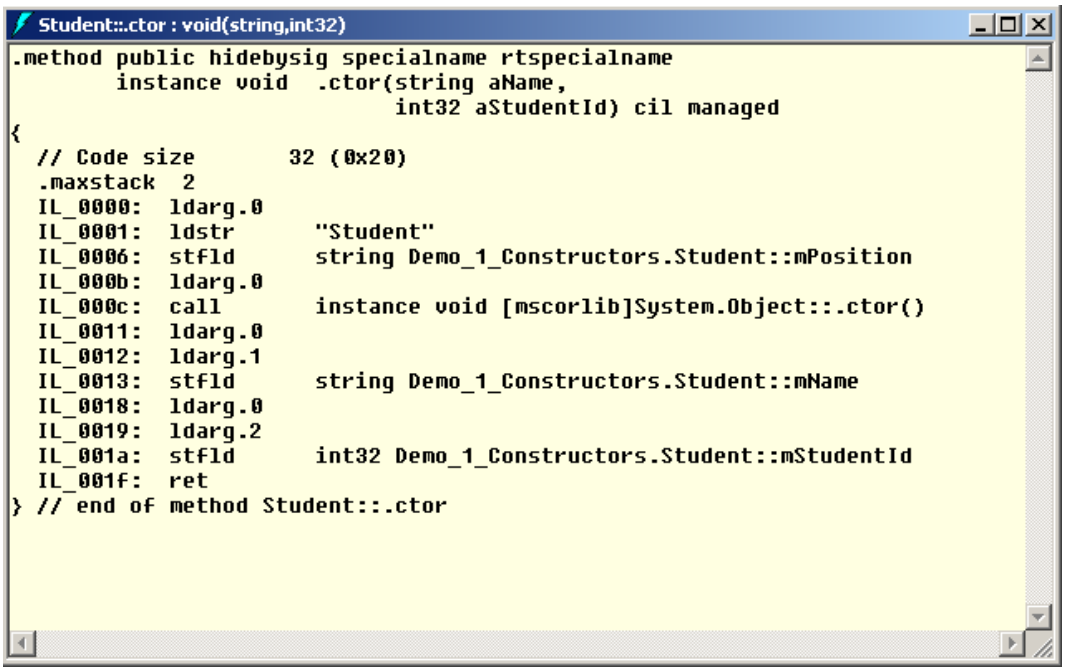


IL DASM показва дърво за асемблито, в което различаваме класа **Student** и членовете му. Ако се придвижим по дървото до конструкторите на класа, можем да изследваме техния IL код, както е показано на следващата картинка:



В кода, генериран за конструктора с един параметър, се вижда обръщението към този с два параметъра. Ако повторим същото действие и с втория конструктор, можем да наблюдаваме и неговия IL код (на картинката по-долу).

Забелязваме, че задаването на стойност на полетата с инициализация при декларацията реално се извършва в началото на втория конструктор. Реално тези полета се инициализират и от първия конструктор, защото той извиква втория.



Singleton клас

В този пример ще представим един популярен шаблон в обектно-ориентирания дизайн – клас, който може да има най-много една инстанция в рамките на цялото приложение. Такъв клас наричаме **singleton**. За реализирането на такива класове се използва следният подход:

```
public sealed class Singleton
{
   private static Singleton mInstance = null;

   private Singleton()
   {
   }

   public static Singleton Instance
   {
      get
      {
         if (mInstance == null)
         {
            mInstance = new Singleton();
         }
         return mInstance;
      }
   }
}
```

Целта на задаването на **private** видимост за конструктора на класа е за да не могат да се създават инстанции освен от членове на класа, както в случая статичното свойство **Instance**. В дефиницията на класа е използвана ключовата дума **sealed**, която указва, че класът не може да бъде наследяван.

Горният пример само демонстрира използването на **sealed** класове и частен конструктор. В реална ситуация при реализацията на **singleton** шаблона трябва да се вземе предвид, че е възможно няколко нишки (threads) едновременно да се опитат да извлекат инстанцията на singleton класа и да се получи нежелано поведение. Затова обикновено реализацията на този шаблон изисква допълнителни усилия за нишково обезопасяване на работата на класа. На работата с нишки ще обърнем специално внимание в темата "Многонишково програмиране и синхронизация".

Статичен конструктор

Конструкторите, подобно на други видове членове на класа, могат да бъдат обявени за статични, с тази особеност че статичният конструктор може да бъде най-много един и не може да приема параметри и модификатори за достъп.

Извикване на статичен конструктор

Статичният конструктор се използва за инициализация на статичните членове и се извиква автоматично. Извикването на статичният конструктор се извършва "зад кулисите" от CLR. Това става по време на изпълнението на програмата и моментът на стартирането му не е точно определен. Това, което е сигурно, е че статичният конструктор е вече извикан когато се създаде първата инстанция на класа или когато се достъпи някой негов статичен член. В рамките на програмата, статичният конструктор може да бъде извикан най-много веднъж.

Статичен конструктор – пример

В следващия пример ще разгледаме класа **SqrtPrecalculated**, който използва статичен конструктор:

```
class SqrtPrecalculated
{
  public const int MAX_VALUE = 10000;
  private static int[] mSqrtValues; // static field

  // Static constructor
  static SqrtPrecalculated()
  {
    mSqrtValues = new int[MAX_VALUE + 1];
    for (int i = 0; i <= MAX_VALUE; i++)
      mSqrtValues[i] = (int) Math.Sqrt(i);
  }
```

```
   // Static method
   public static int GetSqrt(int aValue)
   {
      return mSqrtValues[aValue];
   }

   static void Main()
   {
      Console.WriteLine(GetSqrt(1000));
   }
}
```

Класът **SqrtPrecalculated** служи за бързо изчисляване на корен квадратен. Той предоставя статичния метод **SqrtPrecalculated()**, който връща цялата част на квадратния корен на аргумента си.

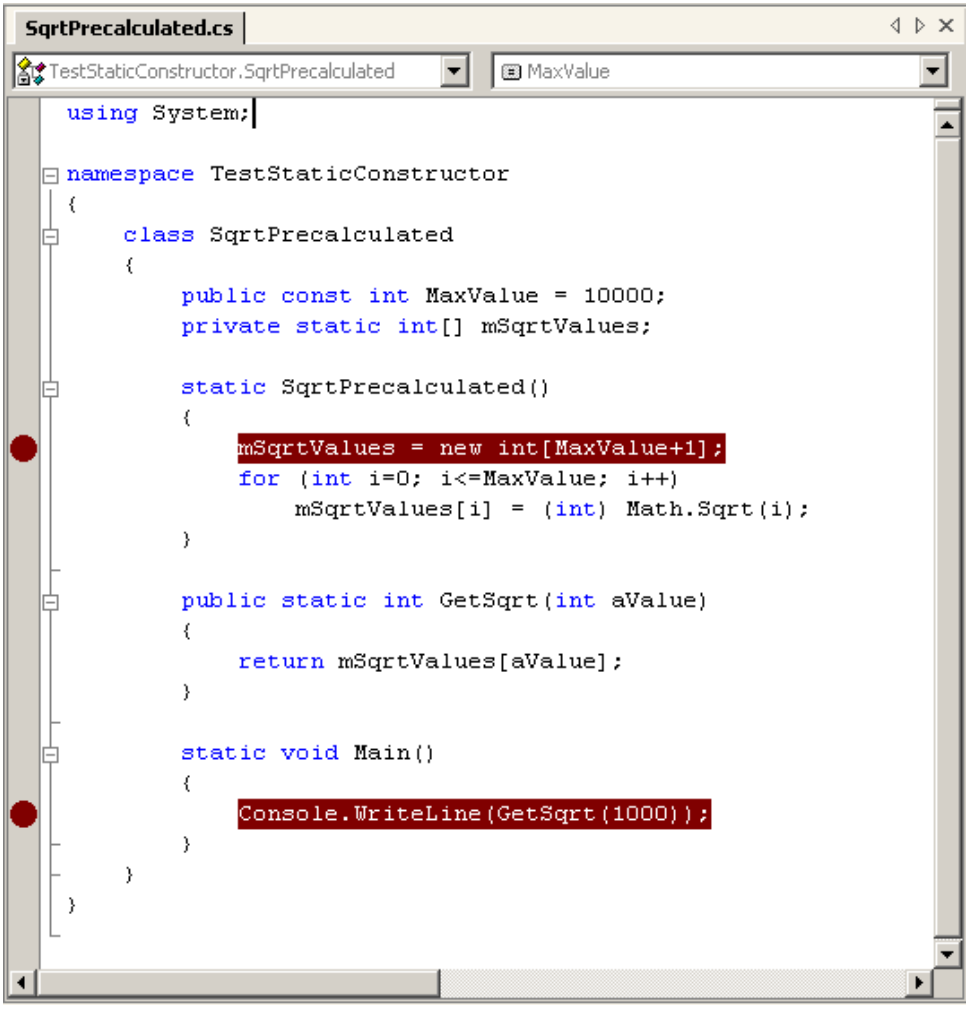
За по-голямо бързодействие всички квадратни корени на числата от 0 до 10000 се изчисляват предварително в статичния конструктор и после се използват наготово. Множеството от стойностите се съхранява в статичното поле **mSqrtValues[]**, което се инициализира в статичния конструктор, който се изпълнява преди първия опит за достъп до класа.

Ще илюстрираме поведението на статичните конструктори в .NET Framework, като с помощта на дебъгера на VS.NET наблюдаваме как преди да започне да бъде използван даден клас се изпълнява първо статичният му конструктор.

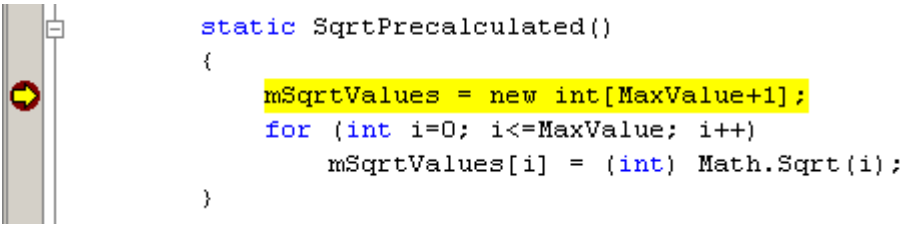
Проследяване на изпълнението на примера

Ще използваме дебъгера на Visual Studio .NET за да проследим изпълнението на кода от горния пример, който се съдържа в приложението **Demo-2-TestStaticConstructor** от демонстрациите. Ще изпълним последователно следните стъпки:

1. Отваряме с VS.NET **TestStaticConstructor.sln** и го компилираме.
2. Слагаме точки на прекъсване (breakpoints) на първия ред на статичния конструктор (**static SqrtPrecalculated()**) и във функцията **Main()** като щракаме с мишката на равнището на тези редове в празното поле от ляво на областта за редактиране на код. След като поставим точките на прекъсване средата изглежда по следния начин:



3. Стартираме програмата в дебъг режим (от меню **`Debug -> Start`** или с [F5]) и проследяваме как дебъгерът на Visual Studio .NET спира първо в статичния конструктор, а след като му зададем да продължи, спира в метода **`Main()`**. Това илюстрира как функционалността от статичния конструктор се изпълнява преди първото използване на класа. Ето как изглежда средата в момента, в който програмата е спряла в статичния конструктор:



Свойства

Свойствата са членове на класовете, структурите и интерфейсите, които обикновено се използват за да контролират достъпа до полетата на типа.

Свойствата приличат на член-променливите по това, че имат име, по-което се достъпват, и стойност от някакъв предварително определен тип. От гледна точка на синтаксиса за достъп до тях, свойствата изглеждат по същият начин както полетата. Разликата се състои в това, че свойства съдържат код, който се изпълнява при обръщение към тях, т. е. извършват действия. Свойствата могат да бъдат и статични.

Прочитане и присвояване на стойност

Свойствата могат да имат два компонента (accessors):

- код за прочитане на стойността (get accessor)
- код за присвояване на стойността (set accessor)

Когато създаваме свойства можем да предоставим дефиниции на двата компонента, както и на само един от тях, но задължително трябва да е дефиниран поне единият. Според предоставените компоненти делим свойствата на три вида:

- Свойства само за чете (read only) - такива, които дефинират само код за прочитане на стойността им.
- Свойства за четене и писане (read and write) - когато имат и двата компонента.
- Свойства само за писане (write only) - когато е предоставен само код за присвояване на стойност.

Пример за свойства

Ще дефинираме класа **`Person`** за да илюстрираме дефинирането и използването на свойства:

```
public class Person
{
  private string mName;
  private DateTime mDateOfBirth;

  // Property Name of type string
  public string Name
  {
     get
     {
        return mName;
     }
     set
     {
```

```
      if ((value != null) && (value.Length > 0))
      {
         mName = value;
      }
      else
      {
         throw new ArgumentException("Invalid name!");
      }
   }
}

// Property DateOfBirth of type DateTime
public DateTime DateOfBirth
{
   get
   {
      return mDateOfBirth;
   }
   set
   {
      if ((value.Year >= 1900) &&
         (value.Year <= DateTime.Now.Year))
      {
         mDateOfBirth = value;
      }
      else
      {
         throw new ArgumentOutOfRangeException(
            "Invalid date of birth!");
      }
   }
}

// Read-only property Age of type int
public int Age
{
   get
   {
      DateTime now = DateTime.Now;
      int yearsOld = now.Year - mDateOfBirth.Year;
      DateTime birthdayThisYear =
         new DateTime(now.Year, mDateOfBirth.Month,
         mDateOfBirth.Day, mDateOfBirth.Hour,
         mDateOfBirth.Minute, mDateOfBirth.Second);
      if (DateTime.Compare(now, birthdayThisYear) < 0)
      {
         yearsOld--;
      }
      return yearsOld;
   }
```

```
   }
}

// Property usage example
class PropertiesDemo
{
   static void Main()
   {
      Person person = new Person();
      person.Name = "Svetlin Nakov";
      person.DateOfBirth = new DateTime(1980, 6, 14);
      Console.WriteLine("{0} is born on {1:dd.MM.yyyy}.",
        person.Name, person.DateOfBirth);
      Console.WriteLine("{0} is {1} years old.",
        person.Name, person.Age);
   }
}
```

В примерния клас виждаме дефинициите на две свойства за четене и писане - **Name** от тип **string** и **DateOfBirth** от тип **DateTime**, както и едно само за четене – **Age** от тип **int**.

Можем да доловим различните аспекти на употребата на свойства - едно свойство може да бъде просто обвивка около поле на типа, но може и да реализира по-сложна логика. Например свойствата **Name** и **DateOfBirth** в примера просто връщат стойността на полетата, които обвиват, или я задават след съответните проверки за валидност. Свойство може да бъде и абстракция на данни, извличането и съхранението на които би могло да бъде свързано със сложна обработка. Опростен пример за това е **Age**, което връща стойност, резултат от извършване на изчисления, в случая разликата между текущата дата и рождената дата на лицето.

Проследяване на изпълнението на свойствата

Ще си изясним работата със свойства като проследим хода на програмата по време на достъпа до тях. За целта ще си послужим с кода от примера, който се съдържа в приложението **Demo-4-Properties** от демонстрациите. Той съдържа горния пример. Нека изпълним следните стъпки:

1. Отваряме с VS.NET **Demo-4-Properties.sln** и го компилираме.
2. Стартираме програмата в режим на проследяване с **[F11]**. Програмата спира изпълнението си на първия ред, но без той да е изпълнен, ето така:

```
class PropertiesDemo
{
    static void Main(string[] args)
    {
        Person person = new Person();
        person.Name = "Svetlin Nakov";
        person.DateOfBirth = new DateTime(1980, 6, 14);

        Console.WriteLine("{0} is born on {1:dd.MM.yyyy}.",
            person.Name, person.DateOfBirth);

        Console.WriteLine("{0} is {1} years old.",
            person.Name, person.Age);
    }
}
```

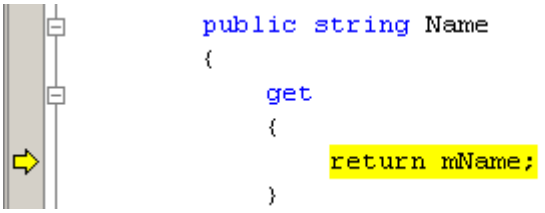
3. Натискаме отново **[F11]** при което се създава обекта **`person`** и маркерът спира на следващия ред.
4. Когато още веднъж натиснем **[F11]**, забелязваме, че кодът, който следва да бъде изпълнен, е тялото на компонента за достъп до свойството **`Name`** на класа **`Person`**:

```
public string Name
{
    get
    {
        return mName;
    }

    set
    {
        if ((value != null) && (value.Length > 0))
        {
```

Това ни показва, че зад операцията "присвояване на стойност" на свойството стои кодът му за присвояване.

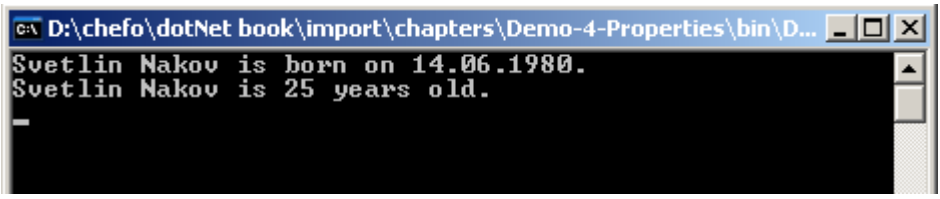
5. С **[Shift-F11]** продължаваме изпълнението на програмата до напускането на текущият блок – то спира отново на следващия ред в тялото на метода **`Main(…)`**.
6. С **[F10]** продължаваме изпълнението на програмата с още една стъпка. Резултатът е същият, както при натискането на **[F11]** с тази разлика, че не се изпълняват стъпка по стъпка вложените блокове. Така преминаваме през изпълнението на кода за присвояване на стойност на свойството **`DateOfBirth`** на "един дъх" и маркерът се позиционира на операцията **`Console.WriteLine("{0} is born on {1:dd.MM.yyyy}.", person.Name, person.DateOfBirth)`**.

7. Ако в този момент натиснем **[F11]** ставаме свидетели на изпълнението и на кода за достъп на свойството **Name**:



Така се убеждаваме, че обръщението към свойство се равнява на изпълнение на кода му за прочитане на стойност.

8. С **[F5]** продължаваме изпълнението на програмата до края и виждаме резултата от изпълнението и:



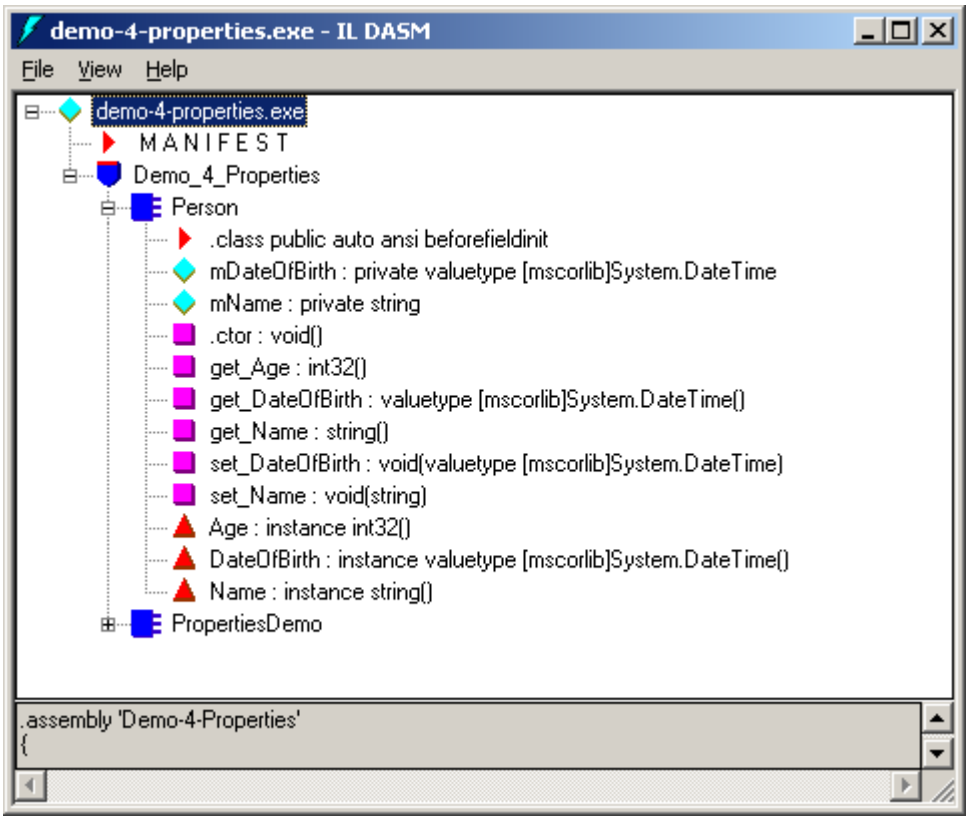
В режим на дебъгване прозорецът, в който се изпълнява приложението, се затваря веднага след приключване на изпълнението на кода и резултатът трудно може да бъде видян. Ако искаме да видим отпечатания резултат, трябва или да сложим точка на прекъсване преди края на `Main()` метода, или да се придвижим до последната операция стъпка по стъпка или да изпълним програмата не с [F5], а с [Ctrl-F5].

9. Нека изследваме с IL DASM генерирания междинен код за примерното приложение, за да си изясним вътрешното представяне на свойствата.

Като стартираме **ildasm** и разгледаме с него IL кода за класа **Person**, забелязваме нещо много интересно – в класа **Person** има методи с префикс **set_**, отговарящи на компонентите за присвояване на дефинираните от нас свойства, и методи с префикс **get_**, които съответстват на компонентите за връщане на стойност.

На практика след компилация **get** и **set** частите на свойствата са се превърнали в методи, а достъпът до тях се е превърнал в операции за извикване на метод. Това е начинът, по който C# компилаторът компилира свойствата – превръща ги в методи, а достъпът до тях превръща в извиквания на методи.

Ето как изглежда класът **Person** в инструмента IL DASM:



Индексатори

Индексаторите в C# (indexers) са членове на класовете, структурите и интерфейсите, които предоставят индексиран достъп до данни на типа, подобно на достъпа до елементите на масив.

Индексаторите по синтаксис и семантика много приличат на свойства, но получават като параметър индекс на елемент, с който да работят. На практика, те представляват свойства, приемащи параметър и дори в някои .NET езици, например VB.NET, синтаксисът на декларирането им е същият като при свойствата.

Индексатори – пример

За да си изясним най-лесно как се дефинират индексатори, да разгледаме следния пример:

```
private object[] mElements;

public object this[int index]
{
   get
   {
```

```
      return mElements[index];
   }
}
```

Виждаме, че дефиницията на индексатор прилича на тази на свойство, но има и някои разлики. На индексатора не се задава име, а вместо него се задава запазената дума **this**.

Достъпът до индексатор на обект се извършва посредством името на променливата от типа, дефиниращ индексатора, последвана от индекса в квадратни скоби, също както се извършва достъпа до елемент на масив, например **myArrayList[5]**.

Позовавайки се на начина, по който се обръщаме към индексаторите, можем да ги разглеждаме като средство за предефиниране на оператора **[]**. Използването на индексатори позволява интуитивен достъп до обекти, които се състоят от множество компоненти, каквито са масивите и колекциите.

Имитация на масив чрез индексатори – пример

За да илюстрираме по-пълно дефинирането и използването на индексатори, ще използваме следващия пример. Ще дефинираме клас, който имитира поведението на масив от 32 стойности, всяка от които е или 0 или 1:

```
struct BitArray32
{
  private uint mValue;

  // Indexer declaration
  public int this [int index]
  {
    get
    {
      if (index >= 0 && index <= 31)
      {
        // Check the bit at position index
        if ((mValue & (1 << index)) == 0)
          return 0;
        else
          return 1;
      }
      else
      {
        throw new ApplicationException(String.
          Format("Index {0} is invalid!", index));
      }
    }
    set
    {
```

```
        if (index < 0 || index > 31)
          throw new ApplicationException(
            String.Format("Index {0} is invalid!", index));

        if (value < 0 || value > 1)
          throw new ApplicationException(
            String.Format("Value {0} is invalid!", value));

        // Clear the bit at position index
        mValue &= ~((uint)(1 << index));

        // Set the bit at position index to value
        mValue |= (uint)(value << index);
      }
    }
}

class IndexerTest
{
  static void Main()
  {
    BitArray32 arr = new BitArray32();

    arr[0] = 1;
    arr[5] = 1;
    arr[5] = 0;
    arr[25] = 1;
    arr[31] = 1;

    for (int i=0; i<=31; i++)
    {
      Console.WriteLine("arr[{0}] = {1}", i, arr[i]);
    }
  }
}
```

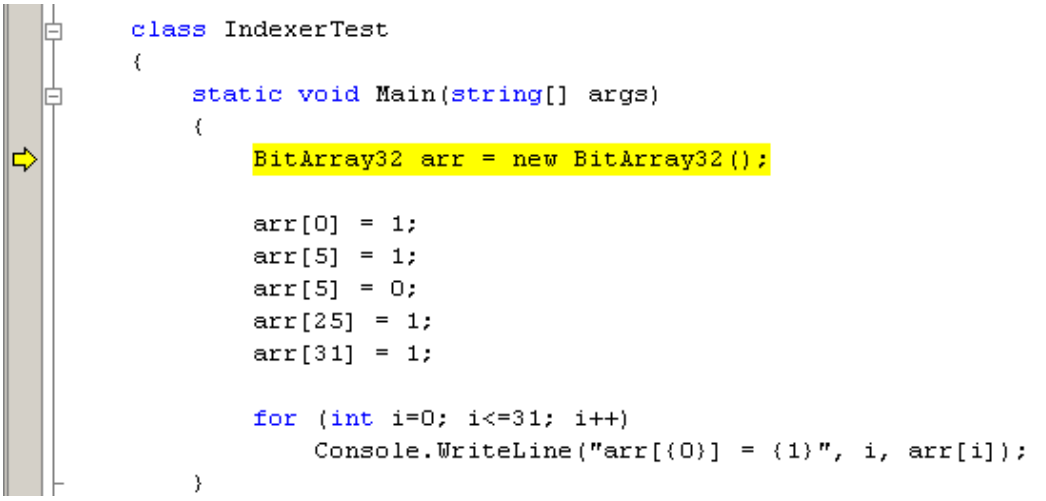
Класът **`BitArray32`** представлява масив от битове с 32 елемента, който вътрешно съхранява стойностите им в едно 32-битово поле. Елементите му достъпваме посредством дефинирания индексатор по същия начин, по който достъпваме елементите на вградените в CTS масиви. На масивите в .NET Framework ще се спрем в темата "Масиви и колекции".

Виждаме компонентите за прочитане и присвояване на стойността, които извършват проверка дали индексът е в съответния диапазон, след което чрез битови операции осъществяват достъп до посочения като параметър бит. При невалидни параметри се предизвиква изключение, чрез което се уведомява извикващия код за проблема. На изключенията ще се спрем подробно в темата "Управление на изключенията в .NET".

Проследяване на работата на индексатор

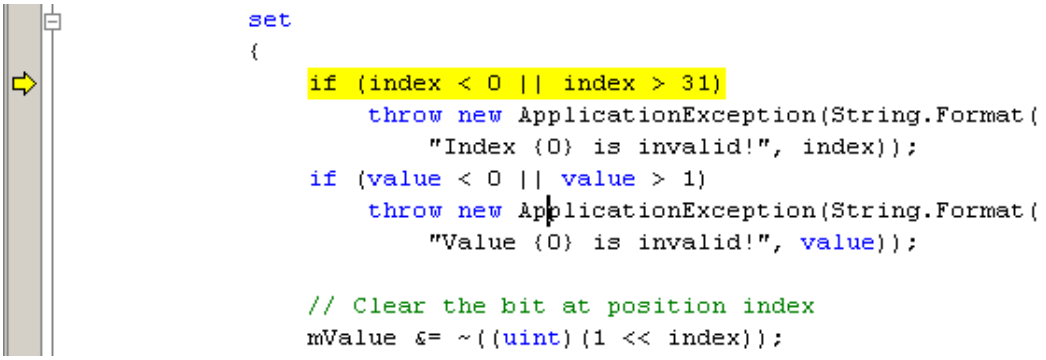
За да проследим работата на индексатора ще си послужим с приложението от демонстрациите **Demo-5-Indexers.sln**, което съдържа кода от горния пример. Ще изпълним следните стъпки:

1. Отваряме приложението и го компилираме.
2. С **[F11]** стартираме изпълнение в режим на проследяване и маркерът се позиционира на първия ред от тялото на метода **Main(…)**:



```
class IndexerTest
{
    static void Main(string[] args)
    {
        BitArray32 arr = new BitArray32();

        arr[0] = 1;
        arr[5] = 1;
        arr[5] = 0;
        arr[25] = 1;
        arr[31] = 1;

        for (int i=0; i<=31; i++)
            Console.WriteLine("arr[{0}] = {1}", i, arr[i]);
    }
```

3. Със следващото натискане на **[F11]** инициализираме обекта **arr** с подразбиращия се конструктор и текущ ред става присвояването **arr[0] = 1**.
4. Когато продължим проследяването, виждаме как следващият код, който се изпълнява, е компонента за присвояване на стойност на индексатора:



```
        set
        {
            if (index < 0 || index > 31)
                throw new ApplicationException(String.Format(
                    "Index {0} is invalid!", index));
            if (value < 0 || value > 1)
                throw new ApplicationException(String.Format(
                    "Value {0} is invalid!", value));

            // Clear the bit at position index
            mValue &= ~((uint)(1 << index));
```

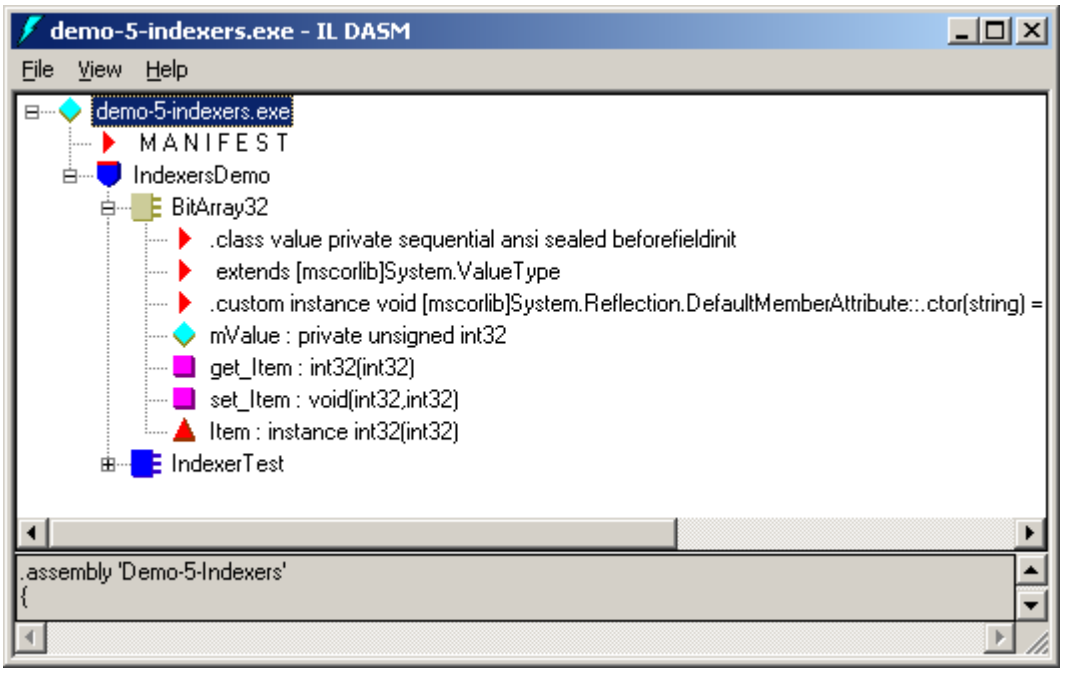
5. С **[Shift-F11]** прескачаме останалата част от блока и преминаваме с **[F10]** през другите присвоявания докато достигнем до цикъла, който прочита стойностите от масива:

```
static void Main(string[] args)
{
    BitArray32 arr = new BitArray32();

    arr[0] = 1;
    arr[5] = 1;
    arr[5] = 0;
    arr[25] = 1;
    arr[31] = 1;

    for (int i=0; i<=31; i++)
        Console.WriteLine("arr[{0}] = {1}", i, arr[i]);
}
```

6. В този момент при натискане на **[F11]** се изпълнява кодът за прочитане на стойността от масива. Забелязваме, че механизмът на изпълнение на индексаторите е същият, както на свойствата.

```
public int this [int index]
{
    get
    {
        if (index >= 0 && index <= 31)
        {
            // Check the bit at position index
            if ((mValue & (1 << index)) == 0)
                return 0;
            else
                return 1;
        }
```

7. С помощта на инструмента IL DASM можем да си обясним приликите между свойства и индексатори. Когато разгледаме генерирания за приложението MSIL код виждаме, че индексаторите, както свойствата, се реализират от двойка методи с имена **`get_Item`** и **`set_Item`**:



Индексатори с няколко параметъра

В .NET Framework се допуска дефинирането на индексатори, приемащи повече от един параметър. Примерно обръщение към такъв индексатор е конструкцията **`personInfo["Бай Иван", 68]`**. Възможно е в един тип да се дефинират и няколко индексатора с различен набор от параметри. Индексаторите не могат да бъдат статични, тъй като реализират индексиране в рамките на дадена инстанция.

Ето още един пример за индексатор, който приема два параметъра от тип символен низ и връща целочислена стойност:

```
class DistanceCalculator
{
  public int this[string aTown1, string aTown2]
  {
    get
    {
      if (aTown1.Equals("София") && aTown2.Equals("Варна"))
        return 470;
      else
        throw new ApplicationException("Unknown distance!");
    }
  }
}

class DistanceTest
{
```

```
   static void Main()
   {
      DistanceCalculator dc = new DistanceCalculator();
      Console.WriteLine("Разстоянието между {0} и {1} е {2} " +
         "километра.", "София", "Варна", dc["София", "Варна"]);
   }
}
```

В примера е реализиран клас, който по дадени имена на два град връща разстоянието между тях. Разбира се, тази функционалност не е реализирана напълно, но целта на примера е да се илюстрира работата с индексатори, а не да се даде завършен проект, който работи.

Структури

Структурите в .NET Framework представляват съвкупност от полета с данни. Те приличат много на класовете, но за разлика то тях са стойностни типове. Инстанциите на структурите имат поведение като примитивните числени типове – разполагат в стека за изпълнение на програмата, предават се по стойност и се унищожават при излизане от обхват.

За разлика от структурите класовете са типове по референция и се разполагат в динамичната памет заради което създаването и унищожаването им е по-бавно. При предаване като параметри се предава само техният адрес в динамичната памет (т. нар. референция).

Структурите, както и класовете, могат да дефинират конструктори, полета, свойства, индексатори и други членове.

Въпреки, че синтаксисът на езика C# го допуска, не се препоръчва в структурите да има методи с логика. Структурите трябва да се използват за да съхраняват някаква структура от данни (съвкупност от полета).

При правилна употреба заместването на класове със структури може значително да увеличи производителността. Ще се спрем по-подробно на класовете и структурите в темата "Обща система от типове".

Структурите се дефинират по същия начин, както и класовете, но вместо запазената дума "**class**" се използва запазената дума "**struct**".

Структури – пример

За да демонстрираме работата със структури, ще дадем няколко примера:

```
struct Point
{
   public int mX, mY;
}

struct Color
{
```

```
   public byte mRedValue;
   public byte mGreenValue;
   public byte mBlueValue;
}

struct Square
{
   public Point mLocation;
   public int mSize;
   public Color mBorderColor;
   public Color mSurfaceColor;
}
```

Както виждаме, структурите много приличат на класове, но основното им предназначение е да съхраняват данни.

Предаване на параметрите

В C# има три различни режима на предаване на параметрите. Ще ги разгледаме накратко, след което ще се спрем по-подробно на всеки от тях и ще илюстрираме разликите между тях с примери. Параметрите при извикване на метод могат да се предават по следните начини:

- **out** (изходни параметри за връщане на стойност)

  Параметрите могат да не бъдат инициализирани преди предаването им. Инициализацията се извършва от извиквания метод, а преди нея достъпът е само за писане и в тялото на метода, и в кода, който го извиква.

- **ref** (входно-изходни параметри за предаване по референция)

  Промените, които методът прави по подадените му по референция параметри, изменят истинските стойности на параметрите, а не техни копия от стека, и за това са видими от кода, извикал метода.

- **in** (входни параметри за предаване по стойност)

  Това е подразбиращият се режим на предаване на параметрите в C#. При изпълнението на метода в стека се записват стойностите на параметрите, с които методът работи и след излизането от тялото му, когато се изтрие върха на стека, промените в параметрите остават изгубени, като стойността на локална променлива, излязла от обхват.

Параметри за връщане на стойност (out)

Предаването на **out** параметри се задейства, като маркираме параметъра с ключовата дума **out**, и в дефиницията на метода, и при извикването му. Целта на тези параметри е не методът да приема като входни данни тяхната стойност, а единствено да я инициализира и да я върне като резултат от изпълнението си. По тази причина те се наричат изходни параметри.

Изходните параметри се предават по адрес в случай на стойностен тип и по адрес на референцията (двоен указател) в случай на референтен тип. Благодарение на това при промяна на стойността им в даден метод тази промяна директно се отразява на променливата, подадена от извикващия метод.

Тъй като върнатата стойност от даден метод в C# може да бъде само една, използвайки **out** параметри можем да върнем на кода, извикал метода, повече стойности. Нека разгледаме следния пример за да илюстрираме връщането на стойност чрез **out** параметри:

```
public struct Point
{
   public int mX, mY;

   public Point(int aX, int aY)
   {
      mX = aX;
      mY = aY;
   }
}

public struct Dimensions
{
   public int mWidth, mHeight;
   public Dimensions(int aWidth, int aHeight)
   {
      mWidth = aWidth;
      mHeight = aHeight;
   }
}

public class Rectangle
{
   private int mX, mY, mWidth, mHeight;

   public Rectangle(int aX,int aY, int aWidth,int aHeight)
   {
      mX = aX;
      mY = aY;
      mWidth = aWidth;
      mHeight = aHeight;
   }

   public void GetLocationAndDimensions(
      out Point aLocation, out Dimensions aDimensions)
   {
      aLocation = new Point(mX, mY);
      aDimensions = new Dimensions(mWidth, mHeight);
   }
```

```
}

class TestOutParameters
{
   static void Main()
   {
      Rectangle rect = new Rectangle(5, 10, 12, 8);

      Point location;
      Dimensions dimensions;

      // location and dimension are not previously initialized
      rect.GetLocationAndDimensions(
         out location, out dimensions);

      Console.WriteLine("({0}, {1}, {2}, {3})",
         location.mX, location.mY,
         dimensions.mWidth, dimensions.mHeight);
      // Result: (5, 10, 12, 8)
   }
}
```

В горния пример са дефинирани структурите **Point** и **Dimensions**, които методът **GetLocationAndDimensions(…)** на класа **Rectangle** използва за да връща чрез изходните си параметри техни инстанции.

Трябва да обърнем внимание на употребата на ключовата дума **out** и на това, че променливите **location** и **dimensions** не са инициализирани никъде в тялото на метода **Main(…)**. Ако параметрите не бяха указани като такива за връщане на стойност, това не би било допустимо – получава се грешка при компилация "Use of unassigned local variable".

Примерът извлича с едно извикване на метод две стойности – местоположението и размерите на даден правоъгълник, като ги записва в инстанции на структурите **Point** и **Dimensions**.

Предаване по референция (ref)

Предаването на параметрите по референция се активира като добавим ключовата дума **ref** към описанието на даден параметър в дефиницията на метода и при извикването му. Такива параметри се наричат входно-изходни.

При предаване на параметри по референция при стартиране на метода в стека не се записват копия на стойностите на параметрите, а указатели към адреса в паметта на оригиналните им стойности. Така извиканият метод може както да чете информация от подадените му параметри, така и да ги изменя и да връща стойности на кода, който го е извикал.

Параметрите поп референция се предават по адрес в случай на стойностен тип и по адрес на референцията (двоен указател) в случай на рефе-

рентен тип. Благодарение на това при промяна на стойността им в даден метод тази промяна директно се отразява на променливата, подадена от извикващия метод. Ще илюстрираме това с пример:

```
public struct Point
{
   internal int mX, mY;

   public static void IncorrectMultiplyBy2(Point aPoint)
   {
      aPoint.mX *= 2; aPoint.mY *= 2;
   }

   public static void MultiplyBy2(ref Point aPoint)
   {
      aPoint.mX *= 2; aPoint.mY *= 2;
   }

   static void Main()
   {
      Point p = new Point();
      p.mX = 5;
      p.mY = -8;
      Console.WriteLine("p=({0},{1})", p.mX, p.mY); // 5,-8
      IncorrectMultiplyBy2(p);
      Console.WriteLine("p=({0},{1})", p.mX, p.mY); // 5,-8
      MultiplyBy2(ref p);
      Console.WriteLine("p=({0},{1})", p.mX, p.mY); // 10,-16
   }
}
```

При изпълнение на примера се вижда, че в тялото на метода **`Main(…)`** не се отразяват промените в предадения в подразбиращия се режим (в случая по стойност) параметър **`p`** при извикването на метода **`IncorrectMultiplyBy2(…)`**. Когато, обаче, параметърът е маркиран като **`ref`**, методът **`MultiplyBy2(…)`** успява да удвои членовете му, тъй като този метод променя директно подадената стойност, а нейно копие.

Предаване по стойност (in)

В подразбиращия се режим при извикване на метод му се подават копия от стойностите на параметрите. Реално стойностните типове се предават по стойност (предава се тяхно копие), а референтните типове се предават по референция (предава се копие на тяхната адреса в динамичната памет, към който сочат).

В по-горния пример видяхме как въпреки промяната в тялото на метода **`IncorrectMultiplyBy2(…)`** предадената по стойност променлива **`p`** не измени реалната си стойност. Трябва да обърнем внимание, че **`p`** е от стойностен тип (инстанция на структурата **`Point`**). Ако **`p`** беше референтен

тип, промените в членовете му щяха да бъдат видими за кода, извикал метода. Защо това е така, въпреки че предаваме параметъра по стойност? На този въпрос ще си отговорим след като разгледаме следващия пример:

```
public class ClassPoint
{
   internal int mX, mY;

   public static void MultiplyBy2(ClassPoint aPoint)
   {
      aPoint.mX *= 2; aPoint.mY *= 2;
   }

   public static void IncorrectErase(ClassPoint aPoint)
   {
      aPoint = null;
   }

   static void Main()
   {
      ClassPoint p = new ClassPoint();
      p.mX = 5;
      p.mY = -8;
      Console.WriteLine("p=({0},{1})", p.mX, p.mY); // 5,-8
      MultiplyBy2(p);
      Console.WriteLine("p=({0},{1})", p.mX, p.mY); // 10,-16
      IncorrectErase(p);
      Console.WriteLine("p=({0},{1})", p.mX, p.mY); // 10,-16
   }
}
```

Забелязваме, че при обръщението към метода **`MultiplyBy2(…)`** дори и без да указваме, че параметърът се предава по референция, полетата на **`p`** успешно се удвояват. Това е така, защото класът **`ClassPoint`** е референтен тип и променливата от този тип представлява указател към паметта, където е записана същинската стойност на обекта.

При извикване на метод с предаване на параметрите по стойност в стека се прави копие на подадената променлива, която в случая е указател (референция) и операциите с това копие изменят реално оригиналната стойност на променливата в динамичната памет.

При промяна на даден параметър, подаден по стойност, например при изменянето на параметъра **`aPoint`** в тялото на метода **`IncorrectErase(…)`**, се изменя единствено копираният на стека указател, а не реалната стойност на обекта. Затова при излизане от метода **`IncorrectErase(…)`**, въпреки че за параметъра **`aPoint`** е зададена стойност **`null`**, променливата **`p`** не е променена и сочи към обекта, който е бил подаден при извикването.

Променлив брой параметри

В C# можем да дефинираме методи с променлив брой параметри. Пример за такъв метод, който неведнъж сме ползвали в нашите примери, е **`Console.WriteLine(…)`**. Предаването на променлив брой параметри в C# се реализира чрез следния синтаксис:

```
static int Sum(params int[] aValues)
{
   int retval = 0;
   foreach(int arg in aValues)
   {
      retval += arg;
   }
   return retval;
}

static void Main()
{
   int sum = Sum(1, 2, 3, 4, 5);
   Console.WriteLine(sum); // The result is 15
}
```

В горния пример дефинирахме метода **`Sum(…)`**, който изчислява сумата на произволен брой цели числа от тип **`int`**. Указваме, че методът приема произволен брой параметри със служебната дума **`params`**. Тя може да се използва най-много веднъж в дефиницията на даден метод и задължително се прилага към последния изреден параметър, който трябва да бъде масив, приемащ множеството от параметрите. На метода от примера могат да бъдат подадени както произволен брой променливи от тип **`int`**, така и масив от тип **`int`**, т. е. допустими извиквания са както **`Sum(1,2,3)`**, така и **`Sum(new Object[]{1,2,3})`**.

Предаване на променлив брой параметри от различен тип

В някои случаи е възможно да се нуждаем от метод, който да приема произволен брой параметри, но не задължително от един и същ тип. Например такъв би могъл да бъде методът, който изчислява сума на произволен брой целочислени параметри, включително и такива, зададени като символни низове. За този метод допустими обръщения биха били **`sum(1, 2, 3)`** както и **`sum(1, "2", 3)`**.

За да реализираме такъв метод, можем да предаваме параметрите чрез масив от по-общ тип, например чрез масив от инстанции на типа **`System.Object`**, който е базов тип за всички типове в .NET Framework. Така можем да работим с произволно множество от параметри, чиито тип различаваме с помощта на оператора за принадлежност към тип **`is`**. В нашия случай за да използваме параметри от тип символен низ, първо ги

конвертираме към желания целочислен тип. Ето примерна реализация на описаната идея:

```
int Sum(params Object[] values)
{
   int retval = 0;
   foreach(Object arg in values)
   {
      if (arg is int)
      {
         retval += (int)arg;
      }
      else if (arg is string)
      {
         retval += int.Parse((string)arg);
      }
   }
   return retval;
}
```

В този вариант методът **`Sum(…)`** извлича целочислената стойност от низа, когато се натъкне на такъв. Забелязваме употребата на оператора **`is`**, който връща **`true`** ако първият му аргумент "е" от типа, подаден като втори аргумент и **`false`** в противен случай.

В примера сме използвали операциите **`(int)arg`** и **`(string)arg`**, които наричаме **преобразуване** на типове. На тях ще се спрем след малко, когато разглеждаме предефинирането на оператори и наследяване.

Предефиниране на оператори

Както и в други обектно-ориентирани езици (например C++), в C# някои оператори могат да бъдат предефинирани. Могат да се предефинират унарни (приемащи един аргумент) и бинарни (приемащи два аргумента) оператори, действащи върху дефинирани от потребителя типове. Предефинирането на оператори се извършва, като разработчикът предоставя собствена имплементация за действието на вградените оператори върху дефинираните от него типове.

Приоритет и асоциативност

Операторите, освен с броя на аргументите си, се характеризират с **приоритет** и **асоциативност**. Когато се съставят изрази, съдържащи прилагане на повече от един оператор, редът на прилагането им се определя от приоритета – операторите се прилагат в реда на намаляване на приоритета им. Нека например разгледаме израза **`a*b+c`**. В този случай умножението ще се извърши преди събирането, тъй като е с по-висок приоритет, т.е. ако **a=1**, **b=2** и **c=5** резултатът ще бъде 7.

Ако имаме израз, който прилага много пъти един и същ оператор, редът на прилагането на тези оператори не може да се определи с помощта на приоритета им. В такъв случай той зависи от асоциативността, която може да бъде лява и дясна. Например, ако имаме израза **`1024 / 128 / 8`**, Резултатът от този израз е **`1`**, тъй като операторът **`/`** е лявоасоциативен, т.е. се прилага от ляво на дясно.

В .NET Framework може да се предефинира действието на операторите върху дефинираните от потребителя типове, но не и техните приоритет и асоциативност.

Операторите в C#

По долу даден е списък с всички оператори в C#, изреден по ред на приоритета им, намаляващ от ляво надясно и отгоре надолу:

- основни: **`(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked`**
- унарни: **`+ - ! ++x --x (T)x`**
- мултипликативни: **`* / %`**
- адитивни: **`+ -`**
- побитови (bitshift): **`<< >>`**
- за сравнение: **`< > <= >= is as`**
- за равенство: **`== !=`**
- логически: **`& ^ |`**
- условни: **`&& || c?x:y`**
- за присвояване: **`= += -= *= /= %= <<= >>= &= ^= |=`**

Предефинируеми оператори

Не всички оператори в C# могат да се предефинират. Предефинируеми са унарните оператори **+**, **-**, **!**, **~**, **++**, **--**, **true** и **false**, бинарните **+**, **-**, *****, **/**, **%**, **&**, **|**, **^**, **<<**, **>>**, **==**, **!=**, **>**, **<**, **>=** и **<=**, и операторите за явно (имплицитно) и неявно (експлицитно) преобразуване на типове.

Дефиницията на предефиниран оператор в C# представлява дефиниция на статичен метод, приемащ един или два параметъра и връщащ някакъв резултат, към който е указана ключовата дума **`operator`**.

Предефиниране на оператори – пример

За да илюстрираме предефинирането на оператори ще дефинираме тип "обикновена дроб" (**`Fraction`**), който съдържа в себе си обикновена дроб (съставена от числител и знаменател). Ще предефинираме всички основни математически операции за работа с обикновени дроби (събиране, изваж-

дане, умножение, деление и т.н.), както и някои други оператори, които улесняват работата с типа **Fraction**. Ето една примерна реализация:

FractionsTest.cs

```
public struct Fraction
{
   private long mNumerator;
   private long mDenominator;

   public Fraction(long aNumerator, long aDenominator)
   {
      // Cancel the fraction and make the denominator positive
      long gcd = GreatestCommonDivisor(aNumerator, aDenominator);
      mNumerator = aNumerator / gcd;
      mDenominator = aDenominator / gcd;

      if (mDenominator < 0)
      {
         mNumerator = -mNumerator;
         mDenominator = -mDenominator;
      }
   }

   private static long GreatestCommonDivisor(
      long aNumber1, long aNumber2)
   {
      aNumber1 = Math.Abs(aNumber1);
      aNumber2 = Math.Abs(aNumber2);
      while (aNumber1 > 0)
      {
         long newNumber1 = aNumber2 % aNumber1;
         aNumber2 = aNumber1;
         aNumber1 = newNumber1;
      }
      return aNumber2;
   }

   public static Fraction operator +(Fraction aF1, Fraction aF2)
   {
      long num = aF1.mNumerator*aF2.mDenominator +
         aF2.mNumerator*aF1.mDenominator;
      long denom = aF1.mDenominator*aF2.mDenominator;
      return new Fraction(num, denom);
   }

   public static Fraction operator -(Fraction aF1, Fraction aF2)
   {
      long num =
         aF1.mNumerator*aF2.mDenominator -
         aF2.mNumerator*aF1.mDenominator;
```

```
    long denom = aF1.mDenominator*aF2.mDenominator;
    return new Fraction(num, denom);
  }

  public static Fraction operator *(Fraction aF1, Fraction aF2)
  {
    long num = aF1.mNumerator*aF2.mNumerator;
    long denom = aF1.mDenominator*aF2.mDenominator;
    return new Fraction(num, denom);
  }

  public static Fraction operator /(Fraction aF1, Fraction aF2)
  {
    long num = aF1.mNumerator*aF2.mDenominator;
    long denom = aF1.mDenominator*aF2.mNumerator;
    return new Fraction(num, denom);
  }

  // Unary minus operator
  public static Fraction operator -(Fraction aFrac)
  {
    long num = -aFrac.mNumerator;
    long denom = aFrac.mDenominator;
    return new Fraction(num, denom);
  }

  // Explicit conversion to double operator
  public static explicit operator double(Fraction aFrac)
  {
    return (double) aFrac.mNumerator / aFrac.mDenominator;
  }

  // Operator ++ (the same for prefix and postfix form)
  public static Fraction operator ++(Fraction aFrac)
  {
    long num = aFrac.mNumerator + aFrac.mDenominator;
    long denom = aFrac.mDenominator;
    return new Fraction(num, denom);
  }

  // Operator -- (the same for prefix and postfix form)
  public static Fraction operator --(Fraction aFrac)
  {
    long num = aFrac.mNumerator - aFrac.mDenominator;
    long denom = aFrac.mDenominator;
    return new Fraction(num, denom);
  }

  public static bool operator true(Fraction aFraction)
  {
```

```
      return aFraction.mNumerator != 0;
   }

   public static bool operator false(Fraction aFraction)
   {
      return aFraction.mNumerator == 0;
   }

   public static implicit operator Fraction(double aValue)
   {
      double num = aValue;
      long denom = 1;
      while (num - Math.Floor(num) > 0)
      {
         num = num * 10;
         denom = denom * 10;
      }
      return new Fraction((long)num, denom);
   }

   public override string ToString()
   {
      if (mDenominator != 0)
      {
         return String.Format("{0}/{1}",
            mNumerator, mDenominator);
      }
      else
      {
         return ("NaN"); // not a number
      }
   }
}

class FractionsTest
{
   static void Main()
   {
      Fraction f1 = (double)1/4;
      Console.WriteLine("f1 = {0}", f1);
      Fraction f2 = (double)7/10;
      Console.WriteLine("f2 = {0}", f2);
      Console.WriteLine("-f1 = {0}", -f1);
      Console.WriteLine("f1 + f2 = {0}", f1 + f2);
      Console.WriteLine("f1 - f2 = {0}", f1 - f2);
      Console.WriteLine("f1 * f2 = {0}", f1 * f2);
      Console.WriteLine("f1 / f2 = {0}", f1 / f2);
      Console.WriteLine("f1 / f2 as double = {0}",
         (double)(f1 / f2));
      Console.WriteLine(
```

```
        "-(f1+f2)*(f1-f2/f1) = {0}", -(f1+f2)*(f1-f2/f1));
    }
}
```

Горният пример дефинира клас, представляващ обвивка на обикновена дроб, или иначе казано, той моделира множеството на рационалните числа. За да могат обектите от типа **Fraction** действително да имат поведение като на числа, той предефинира унарните оператори **-**, **++**, **--**, **true**, **false** и бинарните **+**, **-**, ***** и **/**.

Забелязваме също предефинирането на явно преобразуване от **Fraction** към **double** и имплицитно от **double** към **Fraction**. Добре е да обърнем внимание на това, че вида на преобразуването не е избран случайно. Явно преобразуване се дефинира, когато имаме конвертиране със загуба, тъй като изисква изрично упоменаване на преобразованието. В горния пример конвертирането към **double** е такова, защото някои рационални числа не могат да бъдат представени с плаваща запетая без загуба на точност. Ако дефинираме преобразуването към **double** като имплицитно би било възможно по невнимание да присвоим дроб на число с плаваща запетая, но като изискваме изрично преобразуване компилаторът не допуска потенциално опасната операция. Тъй като конвертирането на число с плаваща запетая към рационално винаги може да се извърши без загуба няма нужда да го определяме като явно.

Виждаме, че е допустимо и предефинирането на операторите **true** и **false**. Това позволява използването на инстанции от тип **Fraction** в булеви изрази. Най-лесно това може да се илюстрира с един прост пример. Нека разгледаме следната модифицирана версия на метода **ToString()** на **Fraction**:

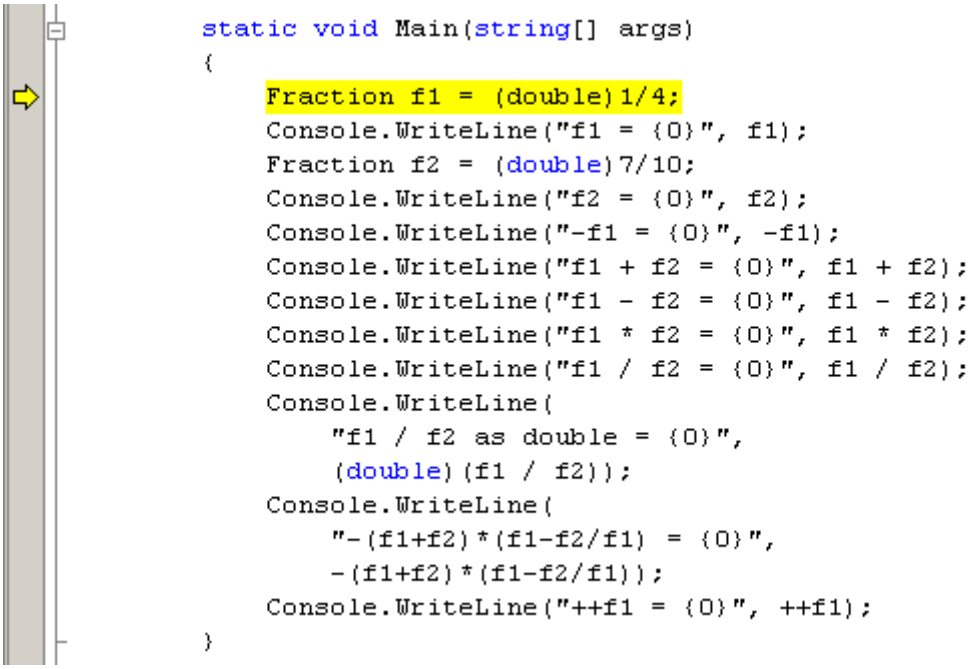
```
public override string ToString()
{
   if (this)
   {
      return String.Format("{0}/{1}", mNumerator, mDenominator);
   }
   else
   {
      return ("0");
   }
}
```

Така променениият метод, освен че връща текстовото представяне на дробта, също и проверява дали тя е нулева дроб и стойност 0 в този случай. При изчисляването на стойността на булевият израз **(this)** се изпълнява тялото на предефинирания оператор **true**.

Проследяване на изпълнението на предефинирани оператори

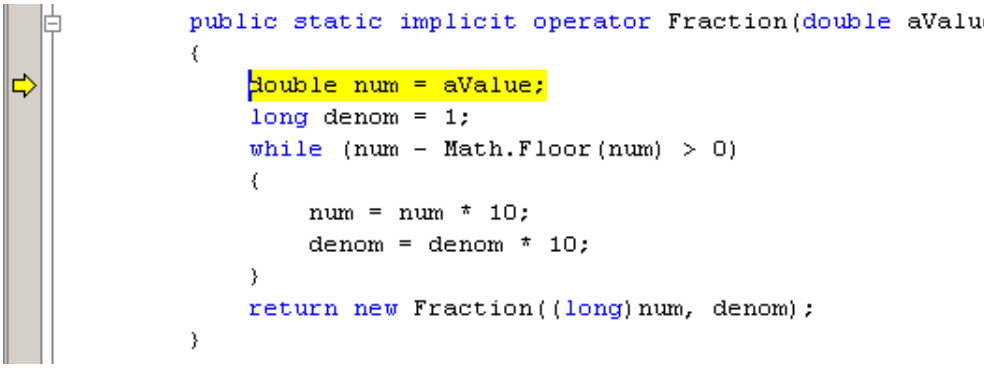
Използвайки дебъгера на Visual Studio .NET ще проследим изпълнението на кода от примера. За целта:

1. Отваряме приложението **Demo-6-Operators.sln** и го компилираме.
2. С **[F11]** стартираме програмата в режим на проследяване и маркерът се позиционира на първия ред, където присвояваме стойност от тип **double** към обекта **f1** от клас **Fraction**:



```
static void Main(string[] args)
{
    Fraction f1 = (double)1/4;
    Console.WriteLine("f1 = {0}", f1);
    Fraction f2 = (double)7/10;
    Console.WriteLine("f2 = {0}", f2);
    Console.WriteLine("-f1 = {0}", -f1);
    Console.WriteLine("f1 + f2 = {0}", f1 + f2);
    Console.WriteLine("f1 - f2 = {0}", f1 - f2);
    Console.WriteLine("f1 * f2 = {0}", f1 * f2);
    Console.WriteLine("f1 / f2 = {0}", f1 / f2);
    Console.WriteLine(
        "f1 / f2 as double = {0}",
        (double)(f1 / f2));
    Console.WriteLine(
        "-(f1+f2)*(f1-f2/f1) = {0}",
        -(f1+f2)*(f1-f2/f1));
    Console.WriteLine("++f1 = {0}", ++f1);
}
```

3. Когато още веднъж натиснем **[F11]**, забелязваме, че при това присвояване по премълчаване се изпълнява предефинираният оператор за имплицитно преобразуване и ходът на изпълнение на програмата продължава в тялото на неговата дефиниция:



```
public static implicit operator Fraction(double aValue)
{
    double num = aValue;
    long denom = 1;
    while (num - Math.Floor(num) > 0)
    {
        num = num * 10;
        denom = denom * 10;
    }
    return new Fraction((long)num, denom);
}
```

4. Продължаваме с **[F11]** да проследяваме изпълнението на програмата и виждаме как при изпълняването на всяка операция с обектите от тип **`Fraction`** се изпълнява кода на съответните предефинирани оператори.

Наследяване

Ще се спрем отново на понятието наследяване поради особената му важност в обектно-ориентираното програмиране. Няма да обясняваме теоретичната страна наследяването, тъй като това е извън обхвата на настоящата тема. Ще обясним само как да извършваме наследяване на класове със средствата на езика C#.

В C# синтаксиса и семантиката на наследяването са близки до тези в други езици за обектно-ориентирани езици, като C++ и Java. За да направим даден клас **`Derived`** наследник на даден друг клас **`Base`**, трябва след декларацията на класа **`Derived`** да сложим двоеточие, следвано от името на класа **`Base`**. За да илюстрираме това, ще разширим един от примерите, които разгледахме по-горе в темата:

```
class Student
{
   private string mName;
   private int mStudentId;
   private string mPosition = "Student";

   public Student(string aName, int aStudentId)
   // ...

   public Student(string aName) : this(aName, -1)
   // ...

   public void PrintName()
   {
      Console.WriteLine("Student name: {0}", mName);
   }
}

public sealed class Kiro : Student
{
   public Kiro() : base("Бай Киро", 12345)
   {
   }

   public void Oversleep()
   {
      //...
   }

   static void Main()
```

```
    {
        Student tosho = new Student("Тошо",  54321);
        Kiro kiro1    = new Kiro();
        Student kiro2 = new Kiro();
        // Kiro kiro3 = new Student("Бай Киро", 12345); // invalid!
        tosho.PrintName();
        kiro1.PrintName();
        // kiro2.Oversleep();
        ((Kiro)kiro2).Oversleep();
    }
}
```

Виждаме, че класът **Kiro** наследява класа **Student**, с което приема от него всички негови полета, свойства, методи и други членове. Разбира се, наследените членове са достъпни за класа **Kiro**, само ако не са били обявени като **private** в базовия клас **Student**.

Трябва да обърнем внимание на третия ред от метода **Main(…)**:

```
Student kiro2 = new Kiro();
```

В него създаваме обект от тип **Kiro**, но го присвояваме на променлива от тип **Student**. Тази операция е напълно коректна, тъй като присвояването на обект от наследен тип в променлива от базов тип е позволено. Обратното, обаче, не е в сила и ако разкоментираме втория ред, приложението не би се компилирало.

Обръщението **kiro1.PrintName()** е също напълно валидно, тъй като класът **Kiro** наследява всички членове на базовия клас **Student** и затова съдържа дефиницията на метода **PrintName()**.

Класове, които не могат да се наследяват (sealed)

В дефиницията на класа **Kiro** забелязваме употребата на ключовата дума **sealed**. С нея указваме, че **Kiro** не може да бъде наследяван от друг клас. Това е пример как чрез забраняването на наследяване можем да създаваме йерархии от класове по-близки до реалните обекти, които представяме. В конкретния пример е удачно да маркираме класа като **sealed**, тъй като той представлява категория, която не може повече да се конкретизира (**Kiro** е клас, който съответства на един конкретен обект от действителността, а не на група различни обекти).

Наследяване при структурите

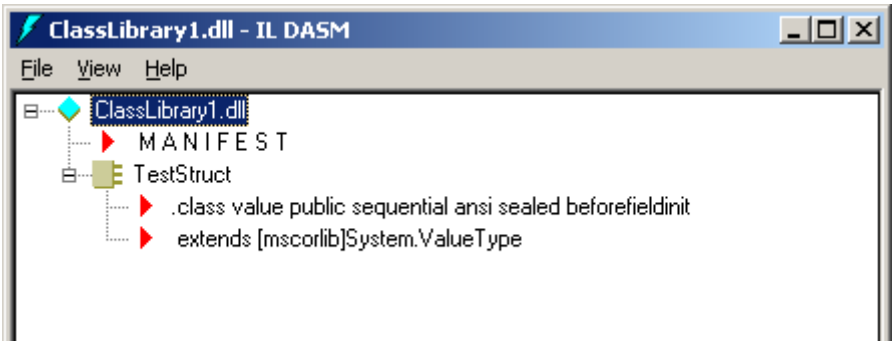
В някои обектно-ориентирани езици, като например C++, се допуска наследяване на структури. В C# и в другите .NET езици това не е позволено.

Структурите в .NET Framework не могат да се наследяват по между си и не могат да наследяват и да бъдат наследявани от класове.

Нека направим един прост експеримент, за да онагледим невъзможността за наследяване на структури. Със следния код ще създадем една тривиална структура:

```
public struct TestStruct
{
}
```

Отново с помощта на инструмента **ildasm** получаваме MSIL кода за тази проста структура:



Забелязваме, че структурата **TestStruct** наследява от **System.ValueType** и, което в нашия случай е по-интересно, в дефиницията ѝ фигурира модификаторът **sealed**. Това указва на компилатора, че този тип не може да бъде наследен. Следната ситуация, при която се опитваме да наследим структура от клас, е също недопустима и предизвиква грешка при опит за компилация:

```
public class TestClass
{
}

public struct AnotherTestStruct : TestClass
{
}
```

Конвертиране на обекти

Нека сега разгледаме конвертирането (casting) на обект от даден тип към обект от друг тип. При класове в отношение наследник-наследен можем да конвертираме нагоре по йерархията (upcasting) и надолу по йерархията (downcasting). Нека обясним тези две понятия.

Конвертиране нагоре (upcasting)

С операцията **`Student kiro2 = new Kiro()`** от по-горния пример присвояваме обект от клас **`Kiro`** на променлива от клас **`Student`**, т.е. **конвертираме** (**преобразуваме**) обекта към класа **`Student`**. В този случай използваме **конвертиране нагоре** (**upcasting**), тъй като **`Student`** е базов клас на **`Kiro`** или, иначе казано, се намира по-горе в йерархията. Тази операция е напълно допустима, тъй като **`kiro2`** действително е студент.

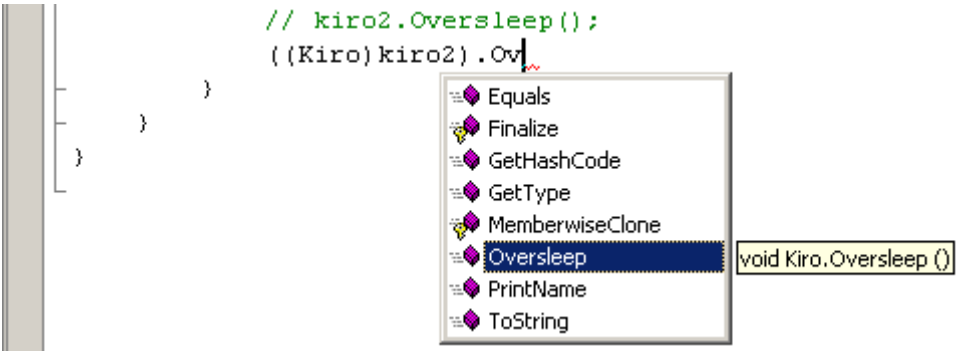
В нашия пример следващият ред

```
Kiro kiro3 = new Student("Бай Киро", 12345);
```

е коментиран, тъй като операцията, която там се опитваме да извършим, е недопустима и този код не би могъл да се компилира, тъй като обектът, който конструираме посредством **`new Student("Бай Киро", 12345)`** не е инстанция на класа **`Kiro`** (въпреки че го наподобява по стойностите на полетата, той не съдържа метода **`Oversleep()`**).

Конвертиране надолу (downcasting)

С обръщението **`(Kiro)kiro2`** разглеждаме обекта **`kiro2`** като обект от тип **`Kiro`**. Тази операция наричаме **конвертиране надолу**, или **downcasting**. Типът на израза в скобите е **`Kiro`** и заради това можем свободно да извикаме метода **`Oversleep()`**, защото въпреки, че е сочен от променлива от тип **`Student`**, този израз фактически е инстанция на класа **`Kiro`** и съдържа имплементация на метода. На долната илюстрация виждаме, че и Visual Studio .NET разпознава типа на израза като ни предоставя членовете му в падащото меню за автоматично завършване на израза:



В C# конвертирането надолу е синтактично валидна операция, независимо дали обектът, който конвертираме, е действително от въпросния наследяващ типа. Например, закоментираното обръщение `Kiro kiro3 = new Student("Бай Киро", 12345)` би могло да се зададе във вида `Kiro kiro3 = (Kiro)new Student("Бай Киро", 12345)`, което се компилира успешно от C# компилатора без дори да генерира преду-

	преждение, тъй като по време на компилация не е известно дали типовете са съвместими. При изпълнението на този код, обаче, въпросното преобразуваме ще предизвика изключение `System.InvalidCastException`, тъй като конструираният обект не е от тип `Kiro` или съвместим с него тип.

Интерфейси

Интерфейсите описват функционалност (група методи, свойства, индексатори и събития), която се поддържа от множество обекти. Подобно на класовете и структурите те се състоят от членове, но се различават от тях по това, че дефинират само прототипите на членовете си, без конкретната им реализацията.

От интерфейсите не могат да се създават обекти чрез инстанциране. Интерфейсите се **реализират** от класове или структури, които имплементират всички дефинирани в тях членове. Конкретните имплементации на даден интерфейс вече могат да се инстанцират и да се присвояват на променливи от тип интерфейс.

Членове на интерфейс

Интерфейсите могат да съдържат методи, свойства, индексатори и събития. В интерфейс не могат да се дефинират конструктори, деструктори, полета и вложени типове и не могат да се предефинират оператори.

Интерфейсите в C# не могат и да съдържат и константи, за разлика от други обектно-ориентирани езици, като Java, където това е допустимо.

Към членовете на интерфейс не може да се прилагат модификатори на достъпа – по подразбиране всички членове са с глобална видимост, все едно е указан модификатор **`public`**. Интерфейс може да наследи един или повече други интерфейса, като е възможно да предефинира или скрива техните членове. За пример да разгледаме няколко дефиниции на интерфейси:

GeometryInterfaces.cs

```
interface IMovable
{
   void Move(int aDeltaX, int aDeltaY);
}

interface IShape
{
   void SetPosition(int aX, int aY);
   double CalculateSurface();
}
```

```
interface IPerimeterShape : IShape
{
   double CalculatePerimeter();
}

interface IResizable
{
   void Resize(int aWeight);
   void Resize(int aWeightX, int aWeightY);
   void ResizeByX(int aWeightX);
   void ResizeByY(int aWeightY);
}

interface IDrawableShape : IShape, IResizable, IMovable
{
   void Delete();

   Color Color
   {
      get;
      set;
   }
}
```

Дефинирахме следните интерфейси: **IMovable**, **IShape**, **IPerimeterShape**, **IResizable** и **IDrawableShape**. Те илюстрират дефинирането на методи и свойства в интерфейс, както и наследяването между интерфейси (което може да бъде и множествено, както е например при **IDrawableShape**).

Реализиране на интерфейс

Тъй като не съдържат данни и описана функционалност, интерфейсите не могат да се инстанцират, а само да се **реализират** (**имплементират**) от класове и структури, от които вече могат да се създават инстанции.

Реализирането на интерфейс е операция, подобна на наследяването, с тази особеност, че реализиращият интерфейса тип в общия случай трябва да предостави реализации за всички членове на интерфейса. Ето примерна реализация на някои от дефинираните в горния пример интерфейси:

GeomertyImplementation.cs

```
public class Square : IShape
{
   private int mX, mY, mSize;

   public Square(int aX, int aY, int aSize)
   {
      mX = aX;
      mY = aY;
```

```
      mSize = aSize;
   }

   public void SetPosition(int aX, int aY) // From IShape
   {
      mX = aX;
      mY = aY;
   }

   public double CalculateSurface() // Derived from IShape
   {
      return mSize * mSize;
   }
}

public struct Rectangle : IShape, IMovable, IResizable
{
   private int mX, mY, mWidth, mHeight;

   public Rectangle(int aX, int aY, int aWidth, int aHeight)
   {
      mX = aX;
      mY = aY;
      mWidth = aWidth;
      mHeight = aHeight;
   }

   public void SetPosition(int aX, int aY) // From IShape
   {
      mX = aX;
      mY = aY;
   }

   public double CalculateSurface() // Derived from IShape
   {
      return mWidth * mHeight;
   }

   public void Move(int aDeltaX, int aDeltaY) // From IMovable
   {
      mX += aDeltaX;
      mY += aDeltaY;
   }

   public void Resize(int aWeight) // Derived from IResizable
   {
      mWidth = mWidth * aWeight;
      mHeight = mHeight * aWeight;
   }
```

```
   public void Resize(int aWeightX, int aWeightY) // IResizable
   {
      mWidth = mWidth * aWeightX;
      mHeight = mHeight * aWeightY;
   }

   public void ResizeByX(int aWeightX) // From IResizable
   {
      mWidth = mWidth * aWeightX;
   }

   public void ResizeByY(int aWeightY) // From IResizable
   {
      mHeight = mHeight * aWeightY;
   }
}

public class Circle : IPerimeterShape
{
   private int mX, mY, mRadius;

   public Circle(int aX, int aY, int aRadius)
   {
      mX = aX;
      mY = aY;
      mRadius = aRadius;
   }

   public void SetPosition(int aX, int aY) // From IShape
   {
      mX = aX;
      mY = aY;
   }

   public double CalculateSurface() // From IShape
   {
      return Math.PI * mRadius * mRadius;
   }

   public double CalculatePerimeter() // From IPerimeterShape
   {
      return 2 * Math.PI * mRadius;
   }
}
```

В този пример виждаме как класът **Square** реализира интерфейса **IShape** и как класът **Rectangle** реализира едновременно няколко интерфейса: **IShape**, **IMovable** и **IResizable**. Класът **Circle** реализира интерфейса **IPerimeterShape**, но понеже този интерфейс е наследник на **IShape**, това означава, че **Circle** на практика имплементира едновременно интерфей-

сите **IShape** и **IPerimeterShape**. Забележете, че всички методи от интерфейсите са декларирани като публични. Това се изисква по спецификация, защото всички методи в даден интерфейс са публични (въпреки, че нямат модификатор **public**). Няма да дискутираме как работят самите имплементации, защото това е извън целите на примера.

Имплементирането на интерфейс много прилича на наследяване. Можем да считаме, че то действително е особен вид наследяване, защото също задава "is-a" релация между интерфейса и типа, който го реализира. Например, в сила са твърденията че квадратът и правоъгълникът са форми, а кръгът също е форма, и освен това има периметър.

След като реализирането на интерфейс създава "is-a" релация, можем да говорим и за множество от обекти от тип интерфейс – това са инстанциите на всички класове, които реализират интерфейса пряко или косвено (реализирайки интерфейс, който го наследява), както и техните наследници.

Реализиране на интерфейс от структура

Интересно в горния пример е, че типът **Rectangle** не е клас, а структура. Това илюстрира една разлика между наследяването на клас и реализирането на интерфейс – второто може да се извърши и от структура.

Въпреки, че е възможно, не е препоръчителна практика структурите да реализират функционалност и да имплементират интерфейси. Структурите трябва да се използват за съхранение на проста съвкупност от полета. Ако случаят не е такъв, трябва да се използва клас.

Обекти от тип интерфейс

Чрез следващия пример ще демонстрираме създаването на обекти от тип интерфейс. Реално ще създаваме обекти от типове, които наследяват даден интерфейс:

GeometryTest.cs

```
class GeomertyTest
{
  public static void Main()
  {
    Square square = new Square(0, 0, 10);
    Rectangle rect = new Rectangle(0, 0, 10, 12);
    Circle circle = new Circle(0, 0, 5);
    if (square is IShape)
    {
      Console.WriteLine("{0} is IShape", square.GetType());
    }
    if (rect is IResizable)
```

```
      {
         Console.WriteLine("{0} is IResizable", rect.GetType());
      }

      IShape[] shapes = {square, rect, circle};
      foreach (IShape shape in shapes)
      {
         shape.SetPosition(5, 5);
         if (shape is IPerimeterShape)
         {
            Console.WriteLine("{0} is IPerimeterShape", shape);
         }
      }
   }
}
```

В горния пример създадохме масив от обекти от тип **IShape** и към всички приложихме действието **SetPosition(…)** полиморфно, т. е. без да се интересуваме от точния им тип – единствено знаем, че обектите поддържат методите от интерфейса. Кодът от примера се компилира и изпълнява без грешка и отпечатва следния резултат:

```
Square is IShape
Rectangle is IResizable
Circle is IPerimeterShape
```

Виждаме, че макар и да не можем директно (с конструктор) да създадем обект от тип интерфейс, можем през променлива от този тип да достъпваме обекти от класовете, които го реализират.

Друго интересно явление, което наблюдаваме в горния пример, е че можем да използваме интерфейс, за да приложим полиморфизъм, като полиморфното действие се извършва от типовете, реализиращи интерфейса, независимо дали са класове или структури.

Запазената дума is

Отново ще обърнем внимание на запазената дума **is**, която представихме при разглеждането на предаването на произволен брой параметри. Обръщението **<обект> is <тип>** връща **true** ако обектът е от дадения тип и **false** в противен случай. Трябва да имаме предвид, че обектите от типнаследник са обекти и от базовия тип, за това **<обект> is <базов_тип>** винаги връща **true**.

В горния пример това обръщение се среща три пъти, като при първите два от тях по време на компилация получаваме предупреждение "The given expression is always of the provided type" – съобщение, с което сме напълно съгласни. Действително, типът на обектите **circle** и **rect** се определя по време на компилация и още тогава е известно, че проверяваното условие е винаги истина.

За обръщението в тялото на цикъла не получаваме предупреждение и в този случай на употреба виждаме истинската мощ на оператора **`is`** – проверка за типа на обект, който не е известен в момента на компилация.

Явна имплементация на интерфейс

Както споменахме по-рано в тази тема, класовете и структурите могат да имплементират по повече от един интерфейс. Това би могло да създаде конфликт, ако един тип имплементира няколко интерфейса, съдържащи методи с еднакви сигнатури. Да разгледаме следния пример:

```
public interface I1
{
   void Test();
}

public interface I2
{
   void Test();
   void AnotherTest();
}

public class TestImplementation : I1, I2
{
   public void Test()
   {
      Console.WriteLine("Test() called");
   }
}
```

Горният код е допустим в C#, но използването му не се препоръчва. То създава затруднения, от една страна, защото не е ясно в кой интерфейс е дефиниран методът **`Test()`** в класа **`TestImplementation`**, и от друга, защото няма възможност да предостави различни имплементации за метода от различните интерфейси.

За да се справим с описания проблем можем да използваме **явната имплементация на интерфейси** (explicit interface implementation). В C# можем да дефинираме в един тип два метода с еднаква сигнатура, стига поне единият от тях да е явна имплементация на метод от интерфейс. Явна имплементация се задава, като изрично се укаже на кой интерфейс принадлежи имплементираният член, както в примера:

```
public class TestExplicit : I1, I2
{
   void I1.Test()
   {
      Console.WriteLine("I1.Test() called");
   }
```

```
  void I2.Test()
  {
    Console.WriteLine("I2.Test called");
  }

  void I2.AnotherTest()
  {
    Console.WriteLine("I2.AnotherTest called");
  }

  public void Test()
  {
    Console.WriteLine("TestExplicit.Test() called");
  }

  public static void Main()
  {
    TestExplicit t = new TestExplicit();

    t.Test();
    // Prints: TestExplicit.Test() called

    I1 i1 = (I1) t;
    i1.Test();
    // Prints: I1.Test() called

    I2 i2 = (I2) t;
    i2.Test();
    // Prints: I2.Test() called
  }
}
```

Виждаме как при явна имплементация на интерфейс трябва да укажем името на интерфейса в дефиницията на реализирания член, а за да го достъпим трябва да преобразуваме обекта към интерфейса. Методите, принадлежащи на явно имплементирани интерфейс, не могат да бъдат публични или да имат друг модификатор за достъп. Те винаги **`private`**.

Не е позволено да имплементираме явно само някои членове от един интерфейс. В горния пример ако променим дефиницията на метода `I2.AnotherTest()` на `public void AnotherTest()`, компилаторът ще съобщи за грешка.

При изпълнение на примерния код се получава следният резултат:

```
TestExplicit.Test() called
I1.Test() called
I2.Test called
```

Абстрактни класове

Абстрактните класове приличат на интерфейсите по това, че те не могат да се инстанцират, защото могат да съдържат дефиниции на неимплементирани методи, но за разлика от интерфейсите могат да съдържат и описани действия. Абстрактният клас реално е комбинация между клас и интерфейс – частично имплементиран клас, който дефинира имплементация за някои от методите си, а други оставя абстрактни, без имплементация.

За пример нека разгледаме следния абстрактен клас:

AbstractTest.cs

```
public abstract class Car
{
   public void Move()
   {
      // Move the car
   }

   abstract public int TopSpeed
   {
      // Retrieve the top speed in Kmph
      get;
   }

   public abstract string BrandName
   {
      get;
   }
}
```

Дефинирахме клас, който реализира само един от членовете си – метода **`Move()`** и дефинира други два, без да ги реализира – свойствата **`BrandName`** и **`TopSpeed`**.

Абстрактните класове, подобно на интерфейсите, ни помагат по-адекватно да моделираме зависимости от реалният свят, защото чрез тях могат да се представят абстрактни същности. В нашия пример невъзможността за инстанциране на класа **`Car`** има смисъл, тъй като и в реалността не можем да имаме кола с неопределена марка.

Абстрактни членове

Ключовата дума **`abstract`** в декларацията на класа го определя като абстрактен. Виждаме, че тя може да се приложи и към член. Абстрактни могат да бъдат методите, свойствата, индексаторите и събитията.

Абстрактните членове не могат да имат имплементация, както и член, който не е абстрактен, не може да бъде оставен без такава.

Ако в един клас е дефиниран абстрактен член, класът задължително трябва да бъде обявен за абстрактен. В противен случай получаваме грешка при компилация. Обратното не е задължително – допустимо е да имаме абстрактен клас, на който всички членове са дефинирани.

Наследяване на абстрактни класове

Тъй като абстрактните класове са класове, те имат същата структура - същият набор от членове (полета, константи, вложени типове и т. н.), същите модификатори на видимостта и дори същите механизми за наследяване, но с някои особености. Нека разширим предходния пример:

AbstractTest.cs

```
public class Trabant : Car
{
   public override int TopSpeed
   {
      get
      {
         return 120;
      }
   }

   public override string BrandName
   {
      get
      {
         return "Trabant";
      }
   }
}

public class Porsche : Car
{
   public override int TopSpeed
   {
      get
      {
         return 250;
      }
   }

   public override string BrandName
   {
      get
```

```
      {
         return "Porsche";
      }
   }
}

public class AbstractTest
{
   static void Main()
   {
      Car[] cars = new Car[] {new Trabant(), new Porsche()};
      foreach (Car car in cars)
      {
         Console.WriteLine("A {0} can go {1} Kmph",
            car.BrandName, car.TopSpeed);
      }
   }
}
```

При изпълнението на този код получаваме следния резултат:

```
A Trabant can go 120 Kmph
A Porsche can go 250 Kmph
```

Виждаме, че въпреки че абстрактният клас не може да се инстанцира директно, обектите от наследяващите го класове могат да се разглеждат като обекти от неговия тип. По показания начин можем да използваме абстрактни базови класове, за да задействаме полиморфизъм, или, казано по-общо, да създадем абстрактен корен на дърво от класове.

В примера ползвахме ключовата дума **override**, с която указваме, че даден метод в класа наследник припокрива (замества) оригиналния наследен метод от базовия си клас. В случая базовия клас не предоставя имплементация за припокритите методи, така че припокриването е задължително. Ще разгледаме ключовата дума **override** и нейното действие след малко. Нека сега продължим с абстрактните класове.

Частично реализиране на абстрактните членове

Възможно е абстрактен клас, съдържащ абстрактни членове, да бъде наследен, без всичките му абстрактни членове да бъдат реализирани. Възможно е също клас, който имплементира абстрактните членове на абстрактния си родител, да дефинира допълнително и свои членове, също абстрактни. В този случай класът-наследник също трябва да бъде деклариран като абстрактен, защото съдържа абстрактни членове.

Тези възможности правят още по-гъвкав инструментариума за създаване на йерархии от класове и моделиране на реалния свят. Ще илюстрираме тази възможност със следното разширение на предходния пример:

AbstractTest.cs

```
abstract public class TurboCar : Car
{
   protected Boolean mTurboEnabled = false;

   public void EnableTurbo()
   {
      mTurboEnabled = true;
   }

   public void DisableTurbo()
   {
      mTurboEnabled = false;
   }
}

public class TrabantTurbo : TurboCar
{
   override public int TopSpeed
   {
      get
      {
         return mTurboEnabled ? 220 : 120;
      }
   }

   override public string BrandName
   {
      get
      {
         return "Trabant Turbo";
      }
   }
}

public class AbstractTest
{
   static void Main()
   {
      TurboCar turboCar = new TrabantTurbo();
      Console.WriteLine("A {0} can go {1} Kmph",
         turboCar.BrandName, turboCar.TopSpeed);

      turboCar.EnableTurbo();
      Console.WriteLine(
         "A {0} can go {1} Kmph with turbo enabled",
         turboCar.BrandName, turboCar.TopSpeed);
   }
}
```

Създадохме класа **TrabantTurbo**, който реализира абстрактните свойства, индиректно наследени от класа **TurboCar**. Класът **TurboCar** е разширение на класа **Car**, който също като него е абстрактен, но предоставя допълнителна функционалност за включване на режим "турбо".

Ако един клас наследи от абстрактен и не предостави дефиниции за всички негови абстрактни членове, той трябва задължително също да бъде обявен за абстрактен.

След изпълнението на примера получаваме следния резултат:

```
A Trabant Turbo can go 120 Kmph
A Trabant Turbo can go 220 Kmph with turbo enabled
```

Виртуални членове

В дефинициите на членовете в горните примери забелязваме употребата на запазената дума **override**. Без нея те не биха могли да бъдат компилирани. Това е така, защото въпросните членове са **виртуални**.

Виртуалните членове са един по-особен вид членове, без които полиморфизмът би бил неосъществим. Тяхната особеност проличава при наследяване – на наследяващите класове се дава възможност вместо изцяло да пресъздадат даден наследен виртуален метод, просто да предоставят своя имплементация на същия. Така, ако работим с обект от наследения клас през референция към базовия, той ще разполага с имплементациите, които наследникът е предоставил. Ще си изясним този механизъм при разглеждането на предефиниране и скриване на виртуални членове.

Виртуални членове се дефинират, като в дефиницията им се укаже ключовата дума **virtual**. Всички абстрактни членове, включително и тези, дефинирани в интерфейсите (и те са абстрактни, тъй като нямат имплементация), са винаги виртуални. Поради тази причина в някои обектно-ориентирани езици за програмиране (например в C++) абстрактните членове се наричат още "чисто виртуални".

Предефиниране и скриване

При дефиниране на виртуален член в тип-наследник, чиято сигнатура съвпада с член, дефиниран в някои от базовите типове, той може или да се предефинира (да му се даде нова имплементация) или да се "скрие".

Когато се използва ключовата дума **override**, се реализира **предефиниране** на виртуалния член, а когато се използва ключовата дума **new** – **скриване**, което е и опцията, която се подразбира когато не се укаже никаква ключова дума.

Когато в наследен клас се предефинира виртуален член на базовия, този член е виртуален и в наследения клас.

Най-лесно ще доловим разликата между скриването и предефинирането на членове, като първо обърнем внимание на следната модификация на по-горния пример. В нея вместо абстрактен сме използвали нормален, конкретен клас, който съдържа дефиниции на свойствата, връщащи подразбиращи се стойности и вместо **override** сме използвали **new**:

NonAbstractTest.cs

```
public class Car
{
   public virtual int TopSpeed
   {
      // Retrieve the top speed in Kmph
      get
      {
         return -1;  // Default value
      }
   }

   public virtual string BrandName
   {
      get
      {
         return "unknown"; // Default value
      }
   }
}

public class Trabant : Car
{
   new public int TopSpeed
   {
      get
      {
         return 120;
      }
   }

   new public string BrandName
   {
      get
      {
         return "Trabant";
      }
   }
}
```

```
public class Porsche : Car
{
   new public int TopSpeed
   {
      get
      {
         return 250;
      }
   }

   new public string BrandName
   {
      get
      {
         return "Porsche";
      }
   }
}

public class NonAbstractTest
{
   static void Main()
   {
      Car[] cars = new Car[] {new Trabant(), new Porsche()};
      foreach (Car car in cars)
      {
         Console.WriteLine("A {0} can go {1} Kmph",
            car.BrandName, car.TopSpeed);
      }
   }
}
```

При изпълнението на този код получаваме следния, донякъде разочароващ, резултат:

```
A unknown can go -1 Kmph
A unknown can go -1 Kmph
```

Причината резултатът да се разминава с очакванията ни е, че при скриването на членовете наследяващият клас не предоставя своята дефиниция на базовия. Така, когато достъпваме обект от наследен клас през референция към обект от базовия, разполагаме само с неговите собствени реализации (на базовия клас). Поради това не можем да използваме полиморфизъм – когато достъпваме обект от базов клас, независимо от специфичният му тип, винаги ще ползваме имплементацията, дефинирана в базовия, т. е. той може приеме само една форма.

Трябва да отбележим, че ако пропуснем запазената дума **new**, поведението на кода ще бъде същото, но ще получим предупреждение от компи-

латора "**The keyword new is required on '<method_name>' because it hides inherited member**".

Ако в горния пример заменим **new** с **override**, ще задействаме механизма на полиморфизма и резултатът ще бъде следния:

```
A Trabant can go 120 Kmph
A Porsche can go 250 Kmph
```

Ако в горния пример пропуснем да обявим членовете **TopSpeed** и **BrandName** като виртуални, ще получим същия разочароващ резултат, както и преди:

```
A unknown can go -1 Kmph
A unknown can go -1 Kmph
```

Виждаме, че при използването на полиморфизъм има много варианти да сбъркаме и да получим неправилно поведение. Затова можем да запомним следното правило:

За да действа полиморфизмът, трябва полиморфният метод в базовия тип да е виртуален (да е обявен като `virtual`, `abstract` или да е член на интерфейс) и в класа наследник да е имплементиран с `override`.

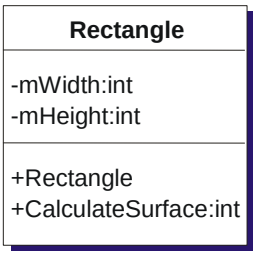
Клас диаграми

Клас диаграмите са стандартно графично средство за изобразяване на йерархии от типове, предоставено ни от езика за моделиране **UML** (Unified Modeling Language). Ще се запознаем съвсем накратко с клас диаграмите без да претендираме за изчерпателност, тъй като моделирането с UML е необятна тема, на която са посветени хиляди страници и тази материя е извън обхвата на настоящата тема.

При многократно наследяване е възможно да се получат йерархии, които са големи и сложни и по тази причина са трудни за възприемане. Чрез клас диаграмите се създава визуална представа за взаимовръзките между типовете и така се улеснява възприемането им. С помощта на клас диаграмите можем да погледнем системата, която разработваме "от птичи поглед", което ни помага да си създадем значително по-ясна представа за нея, отколкото ако преглеждаме множество файлове със сорс код.

Изобразяване на типовете и връзките между тях

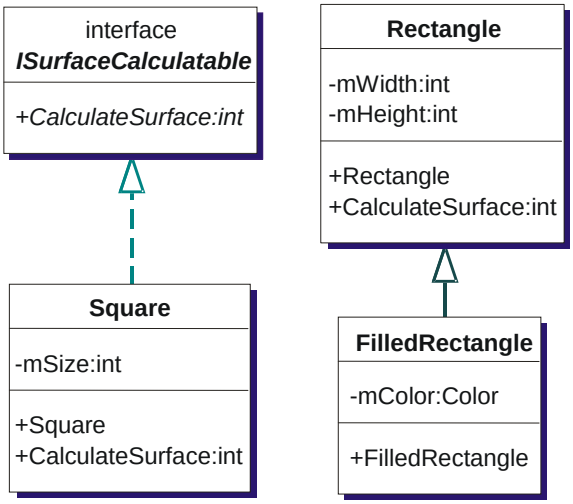
В UML клас диаграмите типовете се изобразяват като правоъгълници, в които са изписани членовете им, евентуално с отбелязана степен на видимост пред името: + за **public**, # за **protected** и - за **private**. Ето един пример (класът **Rectangle**):



Правоъгълникът, изобразяващ даден тип, обикновено е разделен на три части – най-горната съдържа името му, средната съдържа полетата му и най-долната съдържа неговите методи.

Наследяване

Наследяването на клас и имплементирането на интерфейс се изобразява със затворена стрелка (—▷), като стрелките, обозначаващи наследяване и имплементиране се различават по това, че първите обикновено са плътни, а вторите – пунктирани:



В примера класът **`FilledRectangle`** наследява класа **`Rectangle`**, а класът **`Square`** имплементира интерфейса **`ISurfaceCalculatable`**, а.

Асоциация, агрегация, композиция

Връзките между типовете се изобразяват с отворена стрелка (—▷). Тези връзки се наричат още **асоциациационни връзки** (association links).

Асоциационните връзки могат да бъдат три вида (асоциация, агрегация, композиция). **Асоциация** е просто някаква връзка между два типа, примерно даден студент използва даден компютър (асоциацията е между студента и компютъра). **Агрегация** означава че даден клас съдържа много инстанции на даден друг клас, но вторият може да съществува отделно и без първия, примерно една учебна група се състои от много студенти, но студентите могат да съществуват и самостоятелно, без да са

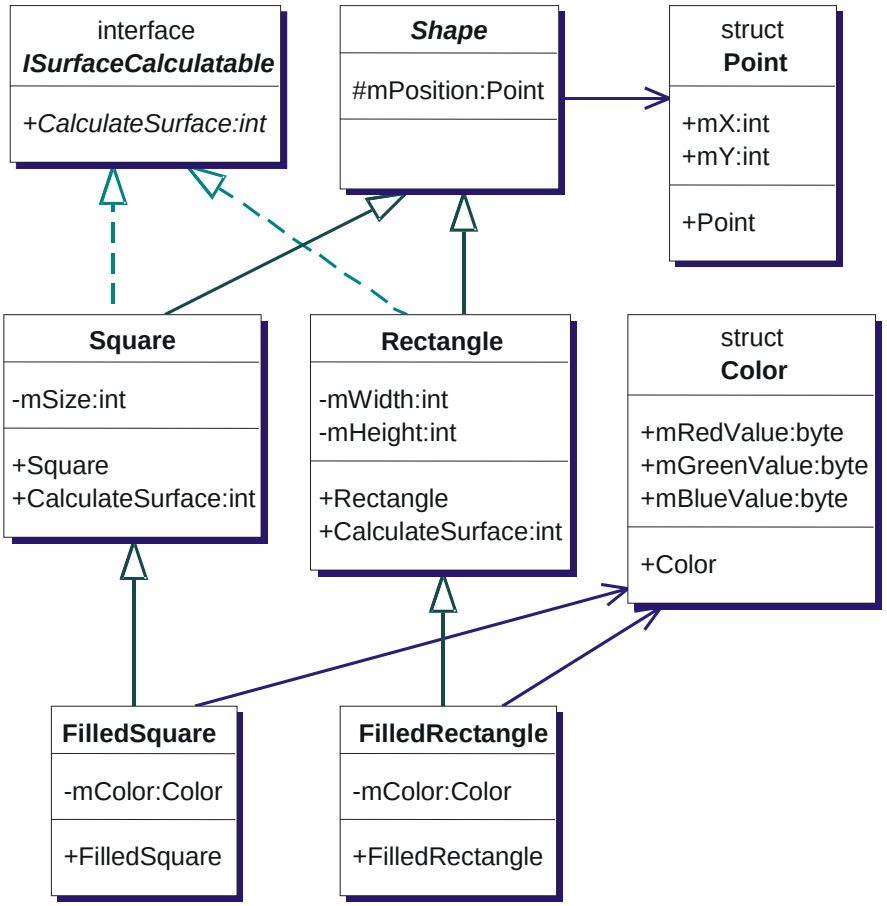
в дадена учебна група. **Композиция** между два класа означава, че един клас се използва като съставна част от друг и не може да съществува без него, примерно един правоъгълник се състои от 4 страни, но страните не могат да съществуват самостоятелно без правоъгълника.

Множественост на връзките

Връзките композиция и агрегация могат да имат **множественост**, например "1 към 1", "1 към много" и т.н. Пример за множественост на връзка е връзката между студент и учебна дисциплина (например "1 към много" – 1 студент изучава много учебни дисциплини).

Клас диаграми – пример

Следният пример представлява проста диаграма и илюстрира основните елементи, които ни предоставя UML нотацията за изграждане на клас диаграми:



По затворените стрелки разбираме, че класовете **`Square`** и **`Rectangle`** наследяват **`Shape`** и имплементират интерфейса **`ISurfaceCalculatable`**, а те

от своя страна са наследени съответно от **FilledSquare** и **FilledRectangle**.

Виждаме също как с отворени стрелки е изобразена връзката "тип съдържа инстанция на друг тип като свой член", както например класът **FilledRectangle** съдържа инстанция на структурата **Color**.

Пространства от имена (namespaces)

Пространствата от имена (namespaces) са средство за организиране на кода в софтуерните проекти. Те съдържат дефиниции на класове, структури, изброени типове и други пространства от имена, като по този начин осигуряват логическо групиране на множества от типове. Пространствата от имена не могат да съдържат дефиниции на функции и данни, тъй като езиците от .NET Framework са строго обектно-ориентирани и такива дефиниции се допускат само в тялото на типовете.

Дефиниране

Пространства от имена в C# се дефинират и използват подобно на пространствата от имена в C++ и на пакетите в Java. Задават се с ключовата дума **namespace** последвана от името на пространството и множеството от дефиниции на типове, оградено във фигурни скоби, както е показано на примера по-долу:

```
namespace SofiaUniversity
{
   // Type definitions ...
}
```

Тази дефиниция може да присъства в повече от един файл, като по този начин се създава пространство, което е физически разпределено в различните файлове.

Достъп до типовете

Достъпът до дефинираните в тялото на пространство типове се осъществява по два начина – чрез използване на **пълно име** на типа и с използването на ключовата дума **using**.

Пълно име на тип

Пълно име наричаме името на типа предшествано от името на пространството, в което се намира, разделени с точка. Например ако класът **AdministrationSystem** е дефиниран в пространството **SofiaUniversity**, тогава пълното му име е **AdministrationSystem.SofiaUniversity**. По този начин се обръщаме към имена на типове, дефинирани в пространства, различни от текущото.

Използването на пространства от имена позволява дефинирането на типове с едно и също име, стига те да са в различни пространства. Посредством използването на пълни имена се разрешават конфликтите, породени от еднаквите имена на типовете. Например клас с име **Config** може да е дефиниран както в пространството **SofiaUniversity.DataAccess**, така и в **SofiaUniversity.InternetUtilities**. Ако е необходимо в даден клас да бъдат използвани едновременно и двата класа, те се достъпват с пълните си имена: **SofiaUniversity.DataAccess.Config** и **SofiaUniversity.InternetUtilities.Config**.

Ключовата дума using

Директивата **using <namespace_name>**, поставена в началото на файла, позволява директно използване на всички типове от указаното пространство само чрез краткото им име. Пример за това е следният фрагмент от кода, който се генерира автоматично от Visual Studio .NET при създаването на нов файл:

```
using System;
```

Това обръщение прави достъпно за програмата основното пространство от имена на .NET Framework – **System**, което съдържа някои типове, които се използват постоянно – **Object**, **String**, **Int32** и др.

Подпространства

Както вече споменахме, пространствата от имена могат да съдържат и дефиниции на други пространства. По този начин можем да създаваме йерархии от пространства от имена, в които да разполагаме типовете, които дефинираме.

Подпространства могат да бъдат дефинирани в тялото на пространството родител, но могат да бъдат създадени и в отделен файл. В такъв случай се използва **пълно име** на пространство от имена. То представлява собственото име на пространството предшествано от родителите му, разделени с точки, както например **System.Windows.Forms**. Пълното име на тип, дефиниран в подпространство, трябва да съдържа пълното му име, например **System.Windows.Forms.Form**.

Следва да илюстрираме дефинирането на една простра структура от пространства от имена:

```
namespace SofiaUniversity.Data
{
   public struct Faculty
   {
      // ...
   }
   public class Student
```

```
    {
        // ...
    }
    public class Professor
    {
        // ...
    }
    public enum Specialty
    {
        // ...
    }
}

namespace SofiaUniversity.UI
{
    public class StudentAdminForm : System.Windows.Forms.Form
    {
        // ...
    }
    public class ProfessorAdminForm : System.Windows.Forms.Form
    {
        // ...
    }
}
namespace SofiaUniversity
{
    public class AdministrationSystem
    {
        public static void Main()
        {
            // ...
        }
    }
}
```

В примера по-горе виждаме дефинициите на основното пространство **SofiaUniversity** и подпространствата му **SofiaUniversity.Data** и **SofiaUniversity.UI**, в които сме дефинирали нашите потребителски типове, например класовете **SofiaUniversity.AdministrationSystem** и **SofiaUniversity.UI.StudentAdminForm**, и структурата **SofiaUniversity.Data.Faculty**.

Използвайки директивата **using** можем да включваме пространства, зададени с пълното им име. Тази директива включва единствено това пространство, което споменаваме изрично, но не и неговите подпространства. Например, ако укажем **using System.Windows** няма да имаме директен достъп до класа **System.Windows.Forms.Form**.

Използвайки ключовата дума **using** можем да задаваме също и псевдоними на пълните имена на пространствата, както например:

```
using WinForms = System.Windows.Forms;

namespace SofiaUniversity.UI
{
   public class StudentAdminForm : WinForms.Form
   {
      // ...
   }

   // ...
}
```

Как да организираме пространствата?

Основната цел на използването на пространства от имена е създаването на добре организирани и структурирани софтуерни системи. За целта трябва да разделяме типовете, които дефинираме, в пространства, чиято структура отговаря на логическата организация на обектите с които работим. Ако се придържаме към някои прости принципи при изграждането на структури от пространства и типове, можем да създадем значително по-ясни и интуитивни за възприемане проекти без да рискуваме вместо това допълнително да си усложним живота.

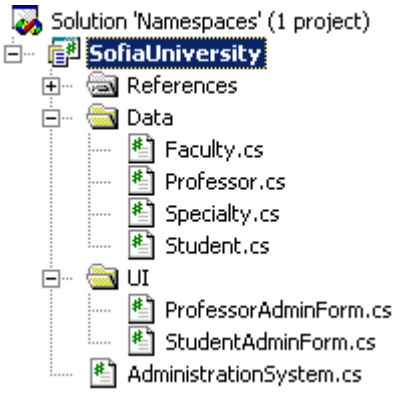
Логическа организация

Изключително полезно е да разпределяме типовете, които дефинираме, в пространства от имена. Това е задължително, ако те са много на брой, например над 20, защото прекалено много елементи на едно място са по-трудни за възприемане не само в програмирането. Можем да създаваме и вложени пространства, но само ако е необходимо - не трябва да изпадаме и в другата крайност, защото ако създаваме прекалено много пространства от имена ще се окажем с излишно сложна структура от пространства, която няма да направи организацията в проекта ни по-ясна, даже напротив.

Физическа организация

Добре е логическата организация в системите, които разработваме, да отговаря на физическата – публичните типове да създаваме във файлове, носещи тяхното име, а за пространствата – директории с тяхното име, в които да се поместват типовете им. Когато създаваме вложени пространства, е добре да ги създаваме като поддиректории на тези на родителите им пространства. Така само с един поглед на структурата на проекта в Solution Explorer на Visual Studio .NET добиваме представа за нея.

За проекта от примера по-горе е удачно да организираме типовете във файлове по следния начин:



Виждаме, че класът **`Student`** от пространството **`SofiaUniversity.Data`** е разположен във файла **`Student.cs`** от поддиректорията **`Data`** на директория **`SofiaUniversity`** от нашия проект. По същия принцип класът **`ProfessorAdminForm`** се намира във файла **`SofiaUniversity/UI/ProfessorAdminForm.cs`**.

При такава организация е много лесно да се запознаем визуално и с логическата, и с физическата структура на компонентите, които изграждат проекта ни. Когато двете не се разминават, навигацията в сорс кода на системата и като цяло работата с нея се улеснява значително.

Принципи при обектно-ориентирания дизайн

Ще разгледаме няколко много важни принципа за ефективно проектиране на типове, които всеки добър софтуерен разработчик трябва да познава и прилага. Тези принципи не се отнасят само за езика C#, а са важни концепции при проектирането и изграждането на софтуер. Те намират приложение дори не само в софтуерното инженерство, но и във всички инженерни дисциплини като цяло.

Когато създаваме софтуерни системи целим да опростим работата по разработването, поддържането и развиването им. Това постигаме като се придържаме към ясна и разбираема структура на системата, близка до проблемната област, към която е ориентирана тази система. Добре направеният обектно-ориентиран дизайн намалява значително усилията за изучаване на системата при извършването на промени. За да го постигнем, е необходимо да се съобразяваме с няколко основни принципа, които ще разгледаме сега.

Функционална независимост (loose coupling)

Когато създаваме типове, които минимално зависят един от друг, можем да променяме всеки от тях без да е необходимо задълбочено познаване на цялата система. Към този принцип за функционална независимост трябва да се придържаме и когато дефинираме членовете на един тип. Ако минимизираме взаимозависимостите в системите, които разработваме, ще

можем много по-лесно да използваме вече създадените модули, типове и методи в други проекти.

При проектирането на типове трябва да следваме принципа, че даден тип трябва да **има ясна цел** и **да зависи минимално от останалите** типове. Тази независимост улеснява поддръжка, опростява дизайна и позволява по-лесно преизползване на кода.

Трябва да се стремим типовете да издават възможно най-малко тайни за това как са имплементирани вътрешно. Потребителите на даден тип трябва да виждат като публични само свойствата и методите, които ги засягат, а останалите трябва да са скрити. Това намалява сложността на системата, защото намалява общия брой детайли, за които потребителят на даден тип трябва да мисли, когато иска да го използва. Скриването на имплементационните детайли (чрез капсулация) позволява промяната в имплементацията на даден тип без да се променя никой от типовете, който го използва.

Тъй като клас диаграмите показват връзките между типовете, те ни помагат да идентифицираме нивото на независимост между тях. Използвайки клас диаграми можем чисто визуално да преценим дали типовете, които използваме, имат прекалено много зависимости помежду си.

Силна логическа свързаност (strong cohesion)

Действията, които даден метод или клас извършва, трябва да бъдат логически свързани, да са **насочени към решаването на една обща задача** (не няколко логически несвързани задачи). Това свойство, е известно още като **модулност**. За да имаме ефективна модуларизация в проекта, който разработваме, трябва всички типове да предоставят ясен интерфейс, който е възможно най-прост и не съдържа излишни методи и свойства. Необходимо е още всички методи в типовете да са свързани логически и да имат имена, които ясно подсказват за какво служат. Не трябва да имаме типове, които имат няколко несвързани логически отговорности и изпълняват разнородни задачи. Това е признак на лош дизайн и води до много проблеми при поддръжката на системата.

Препоръчително е всеки тип, с който работим, както и всеки негов метод, да е свързан с решаването на обща задача и всяко действие, което се извършва да е стъпка или елемент от решаването й. Така системите които изграждаме, ще бъдат много по-разбираеми, и от там лесни за разширяване и поддръжка. Силната свързаност намалява сложността в проектите като спомага за ефективното разделяне на отговорностите в системата.

Упражнения

1. Формулирайте основните принципи на обектно-ориентираното програмиране. Дефинирайте понятията клас, обект, атрибут, метод, енкапсулация на данните, абстракция на данните и действията, наследяване, полиморфизъм.

2. Дефинирайте клас **Student**, който съдържа като **private** полета данните за един студент – трите имена, ЕГН, адрес (постоянен и временен), телефон (стационарен и мобилен), e-mail, курс, специалност, ВУЗ, факултет и т.н. Използвайте изброен тип (enumeration) за специалностите, ВУЗ-овете и факултетите. Дефинирайте свойства за достъп до полетата на класа.
3. Дефинирайте няколко конструктора за класа **Student**, които приемат различни параметри (пълните данни за студента или само част от тях). Неизвестните данни запълвайте с **0** или **null**.
4. Добавете в класа **Student** статично поле, което съдържа количеството инстанции, създадени от този клас от стартирането на програмата до момента. За целта променете по подходящ начин конструкторите на класа, така че да следят броя създадени инстанции.
5. Направете класа **Student** структура. Какви са разликите между клас и структура?
6. Направете нов клас **StudentsTest**, който има статичен метод за отпечатване на информацията за един или няколко студента. Методът трябва да приема променлив брой параметри.
7. Добавете към класа **StudentsTest** няколко статични полета от тип **Student** и статичен конструктор, който създава няколко инстанции на структурата **Student** с някакви примерни данни и ги записва в съответните статични полета.
8. Създайте интерфейс **IAnimal**, който моделира животните от реалния свят. Добавете към него метод **Talk()**, който отпечатва на конзолата специфичен за животното писък, булево свойство **Predator**, което връща дали животното е хищник и булев метод **CouldEat(IAnimal)**, който връща дали животното се храни с посоченото друго животно. За проверка на типа животно използвайте оператора **is**.
9. Създайте класове, които имплементират интерфейса **IAnimal** и моделират животните "куче" и "жаба".
10. Създайте абстрактен клас **Cat** за животното "котка", който имплементира частично интерфейса **IAnimal**.
11. Създайте класове **Kitten** и **Tomcat** за животните "малко котенце" и "стар котарак", които наследяват абстрактния клас **Cat** и имплементират неговите абстрактни методи
12. Създайте клас **CrazyCat**, наследник на класа **Tomcat** за животното "луда котка", което издава кучешки звуци при извикване на виртуалния метод **Talk()**.
13. Реализирайте клас със статичен метод, който инстанцира по един обект от всеки от класовете, поддържащи интерфейса **IAnimal**, и им

извиква виртуалния метод **`Talk()`** през интерфейса **`IAnimal`**. Съответстват ли си животинските писъци на животните, които ги издават?

14. Направете всички полета на структурата **`Student`** с видимост **`private`**. Добавете дефиниции на свойства за четене и писане за всички полета.
15. Направете свойството за достъп до ЕГН полето от структурата **`Student`** само за четене. Направете и полето за ЕГН само за четене. Не забравяйте задължително да го инициализирате от всички конструктори на структурата.
16. Напишете клас, който представя комплексни числа и реализира основните операции с тях. Класът трябва да съдържа като **`private`** полета реална и имагинерна част за комплексното число и да предефинира операторите за събиране, изваждане, умножение и деление. Реализирайте виртуалния метод **`ToString()`** за улеснение при отпечатването на комплексни числа.
17. Реализирайте допълнителни оператори за имплицитно преобразуване на **`double`** в комплексно число и експлицитно преобразуване на комплексно число в **`double`**.
18. Добавете индексатор в класа за комплексни числа, който по индекс **`0`** връща реалната част, а по индекс **`1`** връща имагинерната част на дадено комплексно число.
19. Организирайте всички дефинирани типове в няколко пространства от имена.
20. Направете конструкторите на структурата **`Student`** да подава изключения при некоректно зададени данни за студент.
21. Добавете предизвикване на изключения в класа за комплексни числа, където е необходимо.

Използвана литература

1. Светлин Наков, Обектно-ориентирано програмиране в .NET – http://www.nakov.com/dotnet/lectures/Lecture-3-Object-Oriented-Concepts-v1.0.ppt
2. Jeffrey Richter, Applied Microsoft .NET Framework Programming, Microsoft Press, 2002, ISBN 0735614229
3. Tom Archer, Andrew Whitechapel, Inside C#, 2-nd Edition, Microsoft Press, 2002, ISBN 0735616485
4. Erika Ehrli Cabral, OOPs Concepts in .NET Framework – http://www.c-sharpcorner.com/Code/2005/June/OOPSand.NET1.asp
5. MSDN Training, Programming C# (MOC 2124C), Module 5: Methods and Parameters

6. MSDN Training, Programming C# (MOC 2124C), Module 7: Essentials of Object-Oriented Programming
7. MSDN Training, Programming C# (MOC 2124C), Module 9: Creating and Destroying Objects
8. MSDN Training, Programming C# (MOC 2124C), Module 10: Inheritance in C#
9. MSDN Training, Programming C# (MOC 2124C), Module 12: Operators, Delegates, and Events
10. MSDN Training, Programming C# (MOC 2124C), Module 13: Properties and Indexers
11. MSDN Library – http://msdn.microsoft.com/
12. Visual Case Tool – UML Tutorial, The Class Diagram – http://www.visualcase.com/tutorials/class-diagram.htm
13. Steve McConnell, Code Complete, 2nd Edition, Microsoft Press, 2004, ISBN 0735619670

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 4. Управление на изключенията в .NET

Необходими знания

- Базови познания за архитектурата на .NET Framework
- Базови познания за езика C#

Съдържание

- Какво е изключение в .NET?
- Прихващане
- Свойства
- Йерархия и видове
- Предизвикване (хвърляне)
- Дефиниране на собствени
- Препоръчвани практики

В тази тема ...

В настоящата тема ще разгледаме изключенията в .NET Framework като утвърден механизъм за управление на грешки и непредвидени ситуации. Ще обясним как се прихващат и обработват. Ще разгледаме начините за хвърляне на изключение. Ще се запознаем накратко с различните видове изключения в .NET Framework. Ще дадем примери за дефиниране на собствени (потребителски) изключения.

Изключенията в ООП

В обектно-ориентираното програмиране (ООП) изключенията представляват мощно средство за **централизирана обработка на грешки** и необичайни ситуации. Те заместват в голяма степен процедурно-ориентирания подход, при който всяка функция връща като резултат от изпълнението си код на грешка (или неутрална стойност ако не е настъпила грешка).

В ООП кодът, който извършва дадена операция, обикновено предизвиква изключение, когато в него възникне проблем и операцията не може да бъде изпълнена успешно. Методът, който извиква операцията може да прихване изключението и да обработи грешката или да пропусне изключението и да остави то да бъде прихванато от извикващият го метод. Така не е задължително грешките да бъдат обработвани непосредствено от извикващия код, а могат да се оставят за тези, които са го извикали. Това дава възможност управлението на грешките и необичайните ситуации да се извършва на много нива.

Друга основна концепция при изключенията е тяхната **йерархична същност**. Изключенията в ООП са класове и като такива могат да образуват йерархии посредством наследяване. При прихващането на изключения може да се обработват наведнъж цял клас от грешки, а не само дадена определена грешка (както е в процедурното програмиране).

В ООП се препоръчва чрез изключения да се управлява всяко състояние на грешка или неочаквано поведение, възникнало по време на изпълнението на една програма.

Изключенията в .NET Framework

Изключенията в .NET са класическа имплементация на изключенията от ООП, макар че притежават и допълнителни възможности, произтичащи най-вече от предимствата на управлявания код.

В .NET Framework управлението на грешките се осъществява предимно чрез изключения. Всички операции от стандартната библиотека на .NET (Framework Class Library) сигнализират за грешки посредством **хвърляне** (**throw**, **raise**) на изключение. .NET програмистите трябва да се съобразяват с изключенията, които биха могли да възникнат и да предвидят код за тяхната обработка в някой от извикващите методи.

Изключение може да възникне поради грешка в нашия код или в код който извикваме (примерно библиотечни функции), при изчерпване на ресурс на операционната система, при неочаквано поведение в .NET средата (примерно невъзможност за верификация на даден код) и в много други ситуации.

В повечето случаи едно приложение е възможно да се върне към нормалната си работа след обработка на възникнало изключение, но има и ситуации в които това е невъзможно. Такъв е случаят при възникване на

някои **runtime** изключения. Пример за подобна изключителна ситуация е, когато една програма изчерпа наличната работна памет. Тогава CLR хвърля изключение, което сигнализира за настъпилия проблем, но програмата не може да продължи нормалната си работа и единствено може да запише състоянието на данните, с които работи (за да минимизира загубите), и след това да прекрати изпълнението си.

Всички изключения в .NET Framework са обекти, наследници на класа **`System.Exception`**, който ще разгледаме в детайли след малко. Всъщност, съществуват и изключения, които не отговарят на това изискване, но те са нестандартни и възникват рядко. Тези изключения не са съвместими със CLS (Common Language Specification) и не могат да се предизвикат от .NET езиците (C#, VB.NET и т. н.), но могат да възникнат при изпълнение на неуправляван код.

Изключенията носят в себе си информация за настъпилите грешки или необичайни ситуации. Тази информация може да се извлича от тях и е много полезна за идентифицирането на настъпилия проблем. В .NET Framework изключенията пазят в себе си името на класа и метода, в който е възникнал проблемът, а ако асемблито е компилирано с дебъг информация, изключенията пазят и името на файла и номера на реда от сорс кода, където е възникнал проблемът.

Когато възникне изключение, изпълнението на програмата спира. CLR средата запазва състоянието на стека и търси блока от кода, отговорен за прихващане и обработка на възникналото изключение. Ако не го намери в границите на текущия метод, го търси в извикващия го метод. Ако и в него не го намери, го търси в неговия извикващ и т. н. Ако никой от извикващите методи не прихване изключението, то се прихваща от CLR, който показва на потребителя информация за възникналия проблем.

Изключенията улесняват писането и поддръжката на надежден програмен код, като дават възможност за обработката на проблемните ситуации на много нива. В .NET Framework се позволява хвърляне и прихващане на изключения дори извън границите на текущия процес.

Прихващане на изключения

Работата с изключения включва две основни операции – прихващане на изключения и предизвикване (хвърляне) на изключения. Нека разгледаме първо прихващането на изключения в езика C#.

Програмна конструкция try-catch

В C# изключенията се **прихващат** с програмната конструкция **`try-catch`**:

```
try
{
   // Do some work that can raise an exception
}
```

```
catch (SomeExceptionClass)
{
   // Handle the caught exception
}
```

Кодът, който може да предизвика изключение, се поставя в **try** блока, а кодът, отговорен за обработка му – в **catch** блока.

Catch блокът може да посочи т. нар. **филтър** за прихващане на изключения или да го пропусне. Филтърът представлява име на клас, поставен в скобки като параметър на **catch** оператора. В горния пример филтърът задава прихващане на изключения от класа **SomeExceptionClass** и всички класове, негови наследници. Ако филтърът бъде пропуснат, се прихващат всички изключения, независимо от типа им:

```
try
{
   // Do some work that can raise an exception
}
catch
{
   // Any exception is caught here
}
```

Изразът **catch** може да присъства няколко пъти съответно за различните типове изключения, които трябва да бъдат прихванати, например:

```
try
{
   // Do some work that can raise an exception
}
catch (SomeExceptionClass)
{
   // Handle the SomeExceptionClass and its descendants
}
catch (OtherExceptionClass)
{
   // Handle the OtherExceptionClass and its descendants
}
```

Как CLR търси обработчик за изключенията?

Когато възникне изключение, CLR търси "най-близкия" **catch** блок, който може да обработи типа на възникналото изключение. Първо се претърсва **try-catch** блокът от текущия метод, към който принадлежи изпълняваният в момента код (ако има такъв блок). Последователно се обхождат асоциираните с него **catch** блокове, докато се намери този, чийто филтър съответства на типа на възникналото изключение.

Ако това претърсване пропадне, се извършва същото претърсване за следващия **try-catch** блок, ограждащ текущия (ако има такъв). Този блок може да се намира в текущия метод, в извикващия го метод или в някой от методите, които са извикали него. Ако търсенето отново пропадне, се търси следващия **try-catch** блок и се проверяват неговите филтри дали улавят възникналото изключение. Търсенето продължава докато се намери първият подходящ обработчик на възникналото изключение или се установи, че няма изобщо такъв.

Търсенето може да обходи целия стек на извикване на методите и да не успее да намери **catch** блок, който да обработи изключението. В такъв случай изключението се обработва от CLR (появява се съобщение за грешка).

Прихващане на изключения – пример

Нека разгледаме един прост пример:

```
static void Main()
{
   string s = Console.ReadLine();

   try
   {
      Int32.Parse(s);
      Console.WriteLine("You entered valid Int32 number {0}", s);
   }
   catch (FormatException)
   {
      Console.WriteLine("Invalid integer number!");
   }
   catch (OverflowException)
   {
      Console.WriteLine("Number too big to fit in Int32!");
   }
}
```

В този пример програма очаква да се въведе цяло число. Ако потребителят въведе нещо различно, ще възникне изключение.

Извикването на метода **Int32.Parse(s)** може да предизвика различни изключения и затова е поставено в **try** блок, към който са асоциирани няколко **catch** блока.

Ако вместо число се подаде някаква произволна комбинация от символи, при извикването на метода **Int32.Parse(s)** ще възникне изключението **System.FormatException**, което ще бъде прихванато и обработено от първия **catch** блок.

Ако потребителят въведе число, по-голямо от максималната стойност за типа **System.Int32**, при извикването на **Int32.Parse(s)** ще възникне

System.OverflowException, чиято обработка се извършва от втория **catch** блок.

Всеки **catch** блок е подобен на метод който приема точно един аргумент от определен тип изключение. Този аргумент може да бъде зададен само с типа на изключението, както е в по-горния пример, а може да се зададе и променлива:

```
catch (OverflowException ex)
{
   // Handle the caught exception
}
```

Тук посредством от променливата **ex**, която е инстанция на класа **System.OverflowException**, можем да извлечем допълнителна информация за възникналото изключение.

Прихващане на изключения на нива – пример

Нека сега разгледаме един по-сложен пример за прихващане на изключения – прихващане на изключения на няколко нива:

```
static void Main()
{
   try
   {
      int result = Calc(100000, 100000, 1);
      Console.WriteLine(result);
   }
   catch (ArithmeticException)
   {
      Console.WriteLine("Calculation failed!");
   }
}

static int Calc(int a, int b, int c)
{
   int result;
   try
   {
      checked
      {
         result = a*b/c;
      }
   }
   catch (DivideByZeroException)
   {
      result = -1;
   }
   return result;
```

```
}
```

В този пример изключенията се прихващат на 2 нива – в **try-catch** блок в метода **Calc(…)** и в **try-catch** блок в метода **Main()**, извикващ **Calc(…)**.

Ако методът **Calc(…)** бъде извикан с параметри **(0, 0, 0)**, ще се получи деление на **0** и изключението **DivideByZeroException** ще бъде прихванато и обработено в **try-catch** блока на **Calc(…)** метода и съответно ще се получи стойност -1.

Ако, обаче, методът **Calc(…)** бъде извикан с параметри **(100000, 100000, 1)**, ще се получи препълване на типа **int**, което в **checked** блок ще предизвика **ArithmeticOverflowException**. Това изключение няма да бъде хванато от **catch** филтъра в **Calc(…)** метода и CLR ще провери следващия **catch** филтър. Това е **try-catch** блокът в метода **Main()**, от който е извикан методът **Calc(…)**. CLR ще открие в него е подходящ обработчик за изключението (**catch** филтърът за класа **ArithmeticException**, на който класът **ArithmeticOverflowException** е наследник) и ще го изпълни. Резултатът ще е отпечатване на съобщението "**Calculcation failed!**".

Възможно е по някаква причина в **Calc(…)** метода да възникне изключение, което не е наследник на **ArithmeticException** (например **OutOfMemoryException**). В такъв случай то няма да бъде прихванато от никой от **catch** филтрите и ще се обработи от CLR.

Свойства на изключенията

Изключенията в .NET Framework са обекти. Класът **System.Exception** е базов клас за всички изключения в CLR. Той дефинира свойства, общи за всички .NET изключения, които съдържат информация за настъпилата грешка или необичайна ситуация.

Ето и някои често използвани свойства:

- **Message –** текстово описание на грешката.
- **StackTrace –** текстова визуализация на състоянието на стека в момента на възникване на изключението. Дава информация за това в кой метод в кой файл и на кой ред във файла е възникнало изключението. Имената на файловете и редовете са налични само при компилиране в дебъг режим.
- **InnerException –** изключение, което е причина за възникване на текущото изключение (ако има такова). Например имаме метод който чете от файл и после форматира прочетените данни. Ако по време на четенето възникне изключение то може да бъде прихванато и да се хвърли ново изключение от друг, собствено дефиниран тип, като прихванатото изключение се присвои на свойството **InnerException**. Целта е обработчикът на изключението да получи информация както

за възникналия проблем, така и за неговия първопричинител. Чрез свойството **InnerException** изключенията могат да се свързват във верига, която съдържа последователно всички изключения, които са причинили изключението в нейното начало.

Ще илюстрираме употребата на свойствата с един пример:

```
using System;

class ExceptionsTest
{
  public static void CauseFormatException()
  {
    string s = "an invalid number";
    Int32.Parse(s);
  }

  static void Main(string[] args)
  {
    try
    {
      CauseFormatException();
    }
    catch (FormatException fe)
    {
      Console.Error.WriteLine(
        "Exception caught: {0}\n{1}",
        fe.Message, fe.StackTrace);
    }
  }
}
```

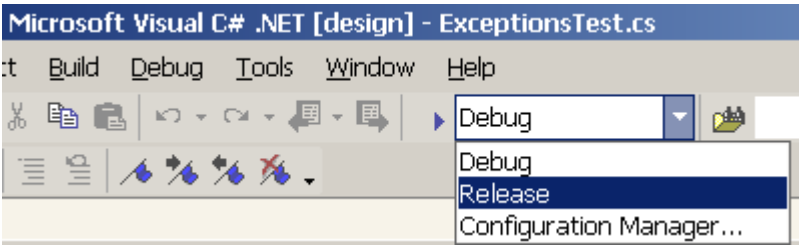
Свойството **StackTrace** е изключително полезно при идентифициране на причината за изключението. Резултатът от примера е информация за прихванатото в **Main()** метода изключение, отпечатана върху стандартния изход за грешки:

```
Exception caught: Input string was not in a correct format.
at System.Number.ParseInt32(String s, NumberStyles style,
NumberFormatInfo info)
at System.Int32.Parse(String s)
at ExceptionsTest.CauseFormatException() in
c:\consoleapplication1\exceptionstest.cs:line 8
at ExceptionsTest.Main(String[] args) in
c:\consoleapplication1\exceptionstest.cs:line 15
```

Имената на файловете и номерата на редовете са достъпни само ако сме компилирали с дебъг информация. Ако компилираме по-горния пример в Release режим, ще получим много по-бедна информация от свойството **StackTrace**:

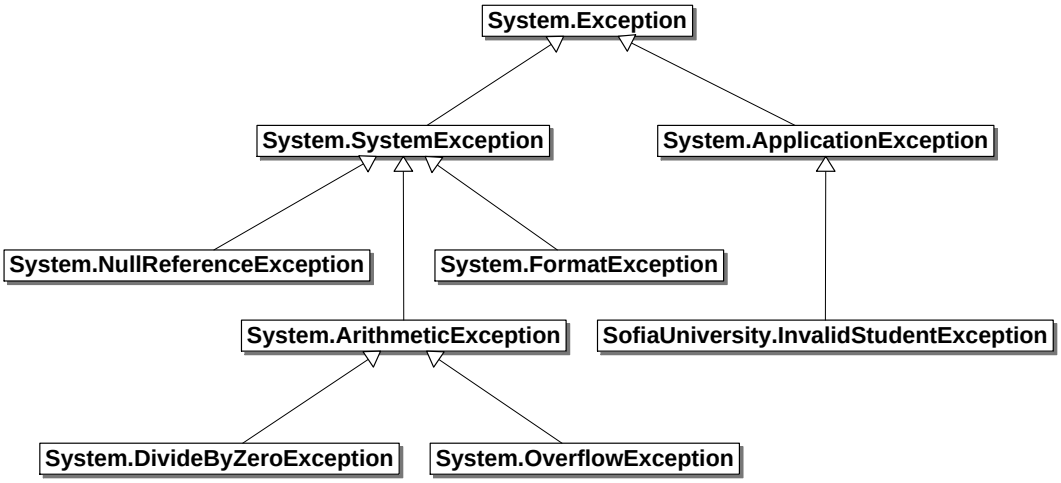
```
Exception caught: Input string was not in a correct format.
at System.Number.ParseInt32(String s, NumberStyles style,
NumberFormatInfo info)
at ExceptionsTest.Main(String[] args)
```

Превключването между Debug и Release на компилация става много лесно от лентата с инструменти за компилация във VS.NET:



Йерархия на изключенията

Изключенията са класове и като такива могат да се наследяват и да образуват йерархии. Както вече знаем, всички изключения в .NET Framework наследяват класа **`System.Exception`**. Този клас има няколко важни наследника, от които обектната йерархия продължава в няколко посоки. Това се вижда от следната диаграма:



Някои изключения директно наследяват **`System.Exception`**, например класовете **`System.SystemException`** и **`System.ApplicationException`**.

Системните изключения, които се използват от стандартните библиотеки на .NET и вътрешно от CLR, наследяват класа **`System.SystemException`**. Ето някои от тях:

- **`System.ArithmeticException`** – грешка при изпълнението на аритметична операция, например деление на 0, препълване на целочислен тип и др.

- **System.ArgumentException** – невалиден аргумент при извикване на метод.
- **System.NullReferenceException** – опит за достъп до обект, който има стойност **null**.
- **System.OutOfMemoryException** – паметта е свършила.
- **System.StackOverflowException** – препълване на стека. Обикновено възниква при настъпване на безкрайна рекурсия.
- **System.IndexOutOfRangeException** – опит за излизане от границите на масив.

Изключенията дефинирани от потребителя трябва да наследяват класа **System.ApplicationException**. Така потребителските програми ще предизвикват изключения само от този тип или негови наследници. Това дава възможност да се разбере дали проблемът е на ниво потребителски код или е свързан със системна грешка. Възможно е потребителски-дефинирано изключение да наследи и директно **System.Exception**, а не **System.ApplicationException**, но има много спорове дали това е добра практика. Някои експерти твърдят, че наследяването на **ApplicationException** усложнява излишно йерархията, докато други смятат, че е по-важно да се разграничават системните от потребителските изключения.

Подредбата на catch блоковете

Както вече знаем, при прихващане на изключения от даден клас се прихващат и изключенията от всички негови наследници. Затова е важна подредбата на **catch** блоковете. Например конструкцията:

```
try
{
   // Do some works that can raise an exception
}
catch (System.ArithmeticException)
{
   // Handle the caught arithmetic exception
}
```

прихваща освен **ArithmeticException** и изключенията **OverflowException** и **DivideByZeroException**. В този пример всичко е наред, но нека разгледаме следния код:

```
static void Main()
{
   string s = Console.ReadLine();
   try
   {
      Int32.Parse(s);
   }
```

```
   catch (Exception) // Трябва да е най-накрая
   {
      Console.WriteLine("Can not parse the number!");
   }
   catch (FormatException) // Този код е недостижим
   {
      Console.WriteLine("Invalid integer number!");
   }
   catch (OverflowException) // Този код е недостижим
   {
      Console.WriteLine("The number is too big!");
   }
}
```

В този пример има недостижим код, защото първият **`catch`** блок ще се изпълнява за всички типове изключения, тъй като той прихваща базовия тип **`System.Exception`**. По тази причина по-специфичните блокове след него няма да се изпълнят никога.

`catch` блоковете трябва да са подредени така, че да започват от изключенията най-ниско в йерархията и да продължават с по-общите. Така ще бъдат обработени първо по-специфичните изключения и след това по общите. В противен случай кодът за по-специфичните никога няма да се изпълни.

Изключения и неуправляван код

Управлявания .NET код може да предизвика само изключения, наследници на **`System.Exception`**. Неуправляваният код може да предизвика и други изключения. За прихващане на всички изключения в C# се използва следната конструкция:

```
try
{
   // Do some work that can raise an exception
}
catch
{
   // Handle the caught exception
}
```

Използването на тази конструкция е опасно и трябва да се използва само в краен случай, когато е наистина е наложително, защото прихващането на всякакви изключения може да доведе до неочаквани резултати.

Предизвикване (хвърляне) на изключения

Досега разгледахме как се прихващат изключения, които са предизвикани от някой друг. Нека сега разгледаме как ние можем да предизвикваме изключения, които някой друг да прихваща.

Предизвикването (**хвърляне**) на изключения (**throwing**, **raising** exceptions) има за цел да уведоми извикващия код за възникването на даден проблем. Тази техника се използва при настъпване на грешка или необичайна ситуация в даден програмен фрагмент. Под "необичайна ситуация" се има предвид ситуация, която разработчикът е предвидил като евентуално възможна, но която не се случва при нормалната работа, примерно опит за намиране на корен квадратен от отрицателно число.

При една такава необичайна ситуация изпълнението на програмата е нормално да продължи, а извикващият текущия метод трябва да бъде информиран за проблема, за да може да реагира по подходящ начин.

За да се хвърли изключение, с което да се уведоми извикващият код за даден проблем в C# се използва оператора **`throw`**, на който се подава инстанция на класа на изключението. Най-често се изисква създаване на обект от някой наследник на класа **`System.Exception`**, в който се поставя описание на възникналия проблем. Ето един пример, в който се хвърля изключение **`ArgumentException`**:

```
throw new ArgumentException("Invalid argument!");
```

Обикновено преди да бъде хвърлено изключение, то се създава чрез извикване на конструктора на класа, на който то принадлежи. Почти всички изключения дефинират следните два конструктора:

```
Exception(string message);
Exception(string message, Exception InnerException);
```

Първият конструктор приема текстово съобщение, което описва възникналият проблем, а вторият приема и изключение, причинител на възникналия проблем.

При хвърляне на изключение CLR прекратява изпълнението на програмата и обхожда стека до достигане на **`catch`** блок за съответното изключение (целият процес беше описан подробно преди малко).

Хвърляне и прихващане на изключения – пример

Ето един пример за хвърляне и прихващане на изключение:

```
public static double Sqrt(double aValue)
{
   if (aValue < 0)
   {
```

```
      throw new System.ArgumentOutOfRangeException(
        "Sqrt for negative numbers is undefined!");
   }
   return Math.Sqrt(aValue);
}

static void Main()
{
   try
   {
      Sqrt(-1);
   }
   catch (ArgumentOutOfRangeException ex)
   {
      Console.Error.WriteLine("Error: " + ex.Message);
   }
}
```

В него е дефиниран метод, който извлича корен квадратен от реално число с двойна точност. При подаване на отрицателен аргумент методът хвърля **`ArgumentException`**. В **`Main()`** метода изключението се прихваща и се отпечатва грешка.

Хвърляне на прихванато изключение – пример

В **`catch`** блокове прихванатите изключения могат да се хвърлят отново. Пример за такова поведение е следният програмен фрагмент:

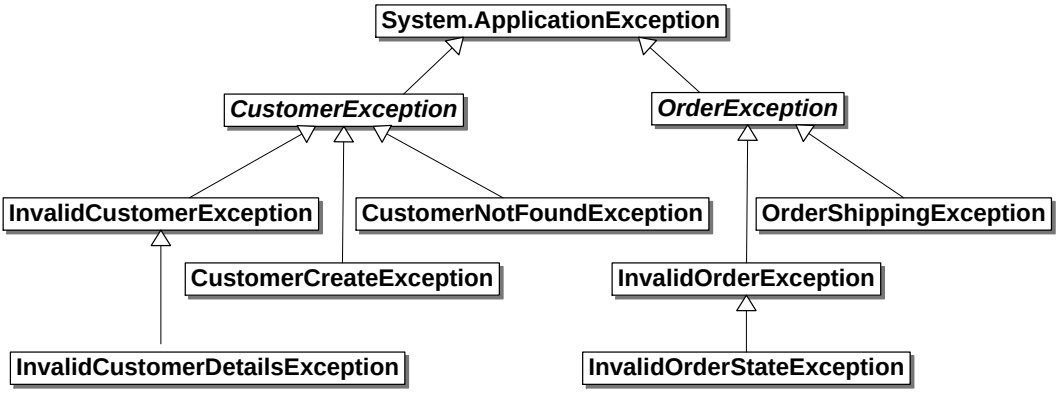
```
public static int Calculate(int a, int b)
{
   try
   {
      return a/b;
   }
   catch (DivideByZeroException)
   {
      Console.WriteLine("Calculation failed!");
      throw;
   }
}

static void Main()
{
   try
   {
      Calculate(1, 0);
   }
   catch (Exception ex)
   {
      Console.WriteLine(ex);
```

```
    }
}
```

В метода **`Calculate(…)`** прихванатото аритметично изключение се обработва като се отпечатва на конзолата "**`Calculation failed!`**" и след това се хвърля отново (чрез израза **`throw;`**). В резултат същото изключение се прихваща и от **`try-catch`** блока в **`Main()`** метода.

Собствени изключения

В .NET Framework програмистите могат да дефинират собствени класове за изключения и да създават класови йерархии с тях. Това осигурява много голяма гъвкавост при управлението на грешки и необичайни ситуации. В по-големите приложения изключенията се разделят в логически в категории и за всяка категория се дефинира по един базов клас, а за конкретните представители на категориите се дефинира по един клас-наследник. Ето един пример:



В примера се създава по един абстрактен базов клас за категорията изключения, свързани с клиентите (**`CustomerException`**) и за категорията изключения, свързани с поръчките (**`OrderException`**). Наследниците на **`OrderException`** и **`CustomerException`** също могат да се подреждат в класова йерархия и да дефинират собствени подкатегории.

При работата на приложението, използващо класовата йерархия от примера могат да се прихващат наведнъж всички грешки, свързани с клиентите или само някои конкретни от тях. Това дава добра гъвкавост при управлението на грешките.

Добре е да се спазва правилото, че йерархиите трябва да са широки и плитки, т.е. класовете на изключения трябва да са производни на тип, който се намира близо до **`System.Exception`**, и трябва да бъдат не повече от две или три нива надълбоко. Ако дефинираме тип за изключение, който няма да бъде базов за други типове, маркираме го като **`sealed`**, а ако не искаме да бъде инстанциран директно, го правим абстрактен.

Дефиниране на собствени изключения

За дефинирането на собствени изключения се наследява класът **System.ApplicationException** и му се създават подходящи конструктори и евентуално му се добавят и допълнителни свойства, даващи специфична информация за проблема. Препоръчва се винаги да се дефинират поне следните два конструктора:

```
MyException(string message);
MyException(string message, Exception InnerException);
```

Въпреки, че не е задължително, силно се препоръчва имената на изключенията да завършват на "**Exception**", например **OrderException**, **CustomerNotFoundException**, **InvalidCredentialsException** и т. н.

Веднъж дефинирани, собствените класове за изключения могат да се ползват по същия начин, както и системните изключения.

Собствени изключения – пример

Ще даден един пример за собствено изключение, което се използва при парсването на текстов файл. То съдържа в себе си специфична информация за проблем, възникнал при парсването – име на файла, номер на ред, съобщение за грешка и изключение-причинител на проблема:

```
class ParseFileException : ApplicationException
{
  private string mFileName;
  private long mLineNumber;

  public string FileName
  {
    get
    {
      return mFileName;
    }
  }

  public long LineNumber
  {
    get
    {
      return mLineNumber;
    }
  }

  public ParseFileException(string aMessage, string aFileName,
    long aLineNumber, Exception aCauseException) : base(
      aMessage, aCauseException)
  {
```

```
      mFileName = aFileName;
      mLineNumber = aLineNumber;
   }

   public ParseFileException(string aMessage, string aFileName,
      Exception aCauseException) :this(
         aMessage, aFileName, 0, aCauseException)
   {
   }

   public ParseFileException(string aMessage, string aFileName) :
      this(aMessage, aFileName, null)
   {
   }
}
```

В класа **ParseFileException** няма нищо сложно. Той наследява **System.Exception** и дефинира две полета (име на файл и номер на ред), две свойства за достъп до тях и няколко конструктора за инициализация на класа по различен набор от параметри.

Понеже всички инстанции на **ParseFileException** се създават чрез извикване (директно или индиректно) на базовия конструктор на класа **ApplicationException**, то при подаване на изключение-причинител, то ще бъде записано в свойството **InnerException**, което се наследява от класа **System.Exception**. По същия начин подаденото текстово описание на проблема ще се запише в наследеното свойство **Message**.

Ето как изключението **ParseFileException** може да бъде използвано в програма, която по даден текстов файл, съдържащ цели числа (по 1 на ред), намира тяхната сума:

```
static long CalculateSumOfLines(string aFileName)
{
   StreamReader inF;
   try
   {
      inF = File.OpenText(aFileName);
   }
   catch (IOException ioe)
   {
      throw new ParseFileException(String.Format(
         "Can not open the file {0} for reading.",
         aFileName), aFileName, ioe);
   }

   try
   {
      long sum = 0;
      long lineNumber = 0;
```

```
      while (true)
      {
         lineNumber++;
         string line;
         try
         {
            line = inF.ReadLine();
         }
         catch (IOException ioe)
         {
            throw new ParseFileException(
               "Error reading from file.",
               aFileName, lineNumber, ioe);
         }

         if (line == null)
            break;  // end of file reached

         try
         {
            sum += Int32.Parse(line);
         }
         catch (SystemException se)
         {
            throw new ParseFileException(String.Format(
               "Error parsing line '{0}'.", line),
               aFileName, lineNumber, se);
         }
      }
      return sum;
   }
   finally
   {
      inF.Close();
   }
}

static void Main()
{
   try
   {
      long sumOfLines = CalculateSumOfLines(@"c:\test.txt");
      Console.WriteLine("The sum of lines={0}", sumOfLines);
   }
   catch (ParseFileException pfe)
   {
      Console.WriteLine("File name: {0}", pfe.FileName);
      Console.WriteLine("Line number: {0}", pfe.LineNumber);
      Console.WriteLine("Exception: {0}", pfe);
   }
```

```
}
```

В кода са използвани класове за работа с текстови файлове и потоци от пространството с имена **System.IO**, които ще разгледаме подробно в темата "Вход и изход". Засега нека се съсредоточим върху използването на изключения, а не върху работата с файлове.

В примера при възникване на проблем при четенето от файла или с формата на данните, прочетени от него, се хвърля изключението **ParseFileException**. В него се задава подходящо съобщение за грешка, записват се името на файла, номерът на реда, където е възникнал проблема, и изключението-причинител на проблема.

Ако стартираме приложението в момент, в който файлът **c:\test.txt** липсва, ще получим следния резултат:

```
File name: c:\test.txt
Line number: 0
Exception: ParseFileException: Can not open the file c:\test.txt
for reading. ---> System.IO.FileNotFoundException: Could not
find file "c:\test.txt". File name: "c:\test.txt"
   at System.IO.__Error.WinIOError(Int32 errorCode, String str)
   at System.IO.FileStream..ctor(String path, FileMode mode,
FileAccess access, FileShare share, Int32 bufferSize, Boolean
useAsync, String msgPath, Boolean bFromProxy)
   at System.IO.FileStream..ctor(String path, FileMode mode,
FileAccess access, FileShare share, Int32 bufferSize)
   at System.IO.StreamReader..ctor(String path, Encoding
encoding, Boolean detectEncodingFromByteOrderMarks, Int32
bufferSize)
   at System.IO.StreamReader..ctor(String path)
   at System.IO.File.OpenText(String path)
   at Test.CalculateSumOfLines(String aFileName) in
c:\demos\ParseFileExceptionDemo.cs:line 52
   --- End of inner exception stack trace ---
   at Test.CalculateSumOfLines(String aFileName) in
c:\demos\ParseFileExceptionDemo.cs:line 56
   at Test.Main() in c:\demos\ParseFileExceptionDemo.cs:line 106
```

Както се вижда, възникнало е изключение **ParseFileException**, а причината за него е изключението **System.IO.FileNotFoundException**.

Съхраняването на началната причина за възникване на изключението при подаване на изключение от по-високо ниво на абстракция (както в горния пример) е добра практика, защото дава на разработчика по-богата информация за възникналия проблем.

В примера изключението **ParseFileException** е от по-високо ниво на абстракция, отколкото **FileNotFoundException** и дава по-богата информация на разработчика.

Конструкцията try–finally

Когато възникне изключение, изпълнението на програмата спира и управлението се предава на най-близкия блок за обработка на изключения. Това означава, че кодът който е между фрагмента, породил изключението и началото на блока за обработка на изключението няма да се изпълни. Да разгледаме следния фрагмент:

```
StreamReader reader = File.OpenText("example.txt");
string fileContents = reader.ReadToEnd();
reader.Close();
```

В него се отваря за четене даден файл, след това се прочита цялото му съдържание и накрая се затваря. Ако по време на четенето от файла настъпи някакъв проблем, последният ред няма да се изпълни и файлът ще остане отворен. Това води до загуба на ресурси и ако се случва често, свободните ресурси малко по малко ще намаляват и в един момент ще се изчерпат. Програмата ще започне да се държи странно и най-вероятно ще приключи работата си аварийно.

Къде е проблемът?

Проблемът е в това, че при възникване на изключение редовете, които следват реда, в който е настъпило изключението, въобще не се изпълняват. Това може да причини лоши последствия като загуба на ресурси, оставяне на обекти в невалидно състояние, неправилен ход на изпълняваните алгоритми и др.

Решението

Проблемът може да бъде решен чрез програмната конструкция **try-finally** в C#:

```
try
{
   // Do some work that can raise an exception
}
finally
{
   // This block will always execute
}
```

Тя осигурява гарантирано изпълнение на зададен програмен блок независимо дали в блока преди него възникне изключение или не. Конструкцията има следното поведение:

- Ако в **try** блока не възникне изключение, след завършването на изпълнението му, се изпълнява веднага след него и **finally** блокът.

- Ако в **try** блока възникне изключение, изпълнението на **try** блока ще се прекъсне и CLR ще започне да търси обработчик за възникналото изключение. В този случай има две възможности:
    - CLR намира обработчик за изключението. Тогава първо ще се изпълни **finally** блокът и едва след това намереният от CLR обработчик.
    - CLR не намира подходящ обработчик. Тогава първо CLR ще обработи изключението (ще даде някакво съобщение за грешка) и след това ще изпълни **finally** блока.

Може да изглежда сложно, но всъщност не е. Важното нещо, което трябва да запомним е, че **finally** блокът се изпълнява винаги, независимо от това какво се е случило в **try** блока. Останалите детайли не са чак толкова важни.

Конструкцията try-catch-finally

Конструкцията **try-finally** може да се комбинира с конструкцията **try-catch**. Така се получава **try-catch-finally** конструкцията, която работи по следния начин:

- Ако в **try** блока не възникне изключение, се изпълняват последователно **try** и **finally** блоковете.
- Ако в **try** блока възникне изключение, което може да се улови от **catch** филтрите на **try-catch-finally** конструкцията, първо се изпълнява съответният **catch** блок, а след него се изпълнява и **finally** блокът.
- Ако в **try** блока възникне изключение, което не отговаря на **catch** филтрите от **try-catch-finally** конструкцията, CLR търси подходящ **catch** филтър в стека за изпълнение на програмата за да обработи изключението. Отново има две възможности:
    - CLR намира обработчик за изключението. Тогава първо ще се изпълни **finally** блокът и едва след това намереният от CLR обработчик.
    - CLR не намира подходящ обработчик. Тогава първо CLR ще обработи изключението (ще даде някакво съобщение за грешка) и след това ще изпълни **finally** блока.

За повече яснота да разгледаме един пример:

```
try
{
   int a = 0;
   int b = 1/a;
}
```

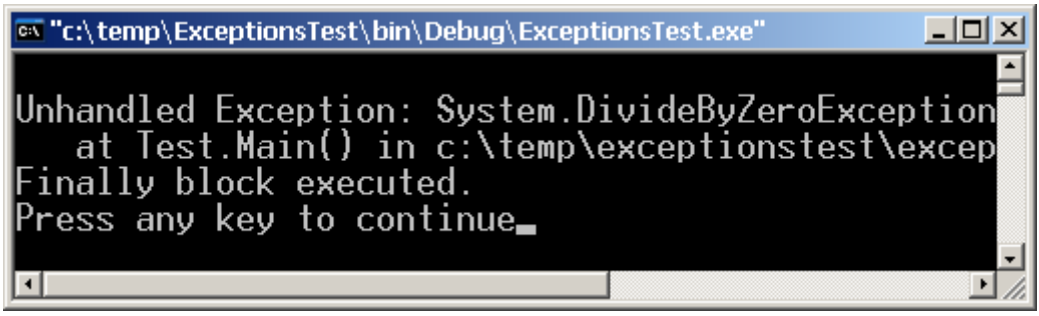
```
catch (ArithmeticException)
{
   Console.WriteLine("ArithmeticException caught.");
}
finally
{
   Console.WriteLine("Finally block executed.");
}
```

В примера в **try** блока възниква аритметично изключение заради делението на 0, то се обработва веднага от **catch** блока и накрая се изпълнява **finally** блокът. При изпълнението на примера се получава следният резултат:

Нека сега разгледаме пример, в който възниква изключение, което не се прихваща никъде в програмата:

```
try
{
   int a = 0;
   int b = 1/a;
}
finally
{
   Console.WriteLine("Finally block executed.");
}
```

В случая CLR вътрешно ще прихване изключението, ще отпечата съобщение за грешка и едва след това ще изпълни **finally** блока:



Да разгледаме и още един пример:

```
try
{
   try
   {
      int a = 0;
      int b = 1/a;
   }
   finally
   {
      Console.WriteLine("Finally block executed.");
   }
}
catch (ArithmeticException)
{
   Console.WriteLine("ArithmeticException caught.");
}
```

При неговото изпълнение ще настъпи аритметично изключение в **try** блока от **try-finally** конструкцията. CLR ще потърси и ще намери подходящ обработчик за него в **try-catch** конструкцията. Понеже CLR е намерил обработчик, първо ще бъде изпълнен **finally** блока, а след това обработчикът на изключението. Резултатът ще бъде следния:

try-finally за освобождаване на ресурси

В блока, който се изпълнява задължително (**finally** блока), може да се съдържа код за освобождаване на ресурси, който трябва да се изпълни винаги. Така се осигурява почистване след всяка успешно започната операция, преди да се върне управлението на извикващия блок или да продължи да се изпълнява кодът след **finally** блока. Ето един пример:

```
/// <summary>
/// Returns the # of first line from the given text file
/// that contains given pattern or -1 if such line is not found
/// </summary>
static int FindInTextFile(string aPattern, string aFileName)
{
   int lineNumber = 0;
   StreamReader inF = File.OpenText(aFileName);
   try
   {
```

```
      while (true)
      {
         string line = inF.ReadLine();
         if (line == null)
           break;  // end of file reached
         lineNumber++;
         if (line.IndexOf(aPattern) != -1)
           return lineNumber;
      }
      return -1;
   }
   finally
   {
      inF.Close(); // The file will never remain opened
   }
}
```

В примера е реализиран метод, който търси даден текст в даден текстов файл и връща номера на реда, в който е намерен текстът. Понеже е използвана конструкцията **try-finally** след отварянето на файла, каквото и да се случи по време на търсенето, файлът накрая ще бъде затворен.

Ако не възникне изключение по време на търсенето, след изпълнението на **return** оператора, ще бъде изпълнен **finally** блокът.

Ако при работата с файла възникне изключение, ще се изпълни първо **finally** блокът и методът няма да върне стойност, а ще завърши с изключение, което ще бъде обработено след това.

Ако при търсенето възникне изключение, но то не бъде прихванато, то ще се обработи от CLR и **finally** блокът ще бъде изпълнен едва след това. Методът няма да върне стойност и програмата ще приключи аварийно.

Препоръчвани практики

Изключенията са много мощен механизъм за обработка на грешки, но ако се използват неправилно, могат да доведат до много трудни за откриване проблеми. Затова ще посочим някои препоръчвани практики при работата с изключения:

- **catch** блоковете трябва да са подредени така, че да започват от изключенията най-ниско в йерархията и да продължават с по-общите. Така ще бъдат обработени първо по-специфичните изключения и след това по общите. В противен случай кодът за по-специфичните никога няма да се изпълни.
- Всеки **catch** блок трябва да прихваща само изключенията, които очаква (и знае как да обработва), а не всички. Лоша практика е да се прихващат всички изключения тъй като различните видове изключения изискват различна обработка и специфични действия за

справяне с възникналата проблемна ситуация. Избягвайте конструкциите **`catch (Exception) {…}`** или просто **`catch {…}`**.

- При дефиниране на собствени изключения трябва да се наследява **`System.ApplicationException`**, а не директно **`System.Exception`**. По този начин може да се направи разграничение на това дали изключението е от .NET Framework или е от приложението.
- Имената на класовете на всички изключения трябва завършват на **`Exception`**, например **`OrderException`**, **`InvalidAccountException`** и т. н. Това прави кода по-разбираем и по-лесен за поддръжка.
- При създаване на инстанция на изключение винаги трябва да й се подава в конструктора подходящо съобщение. Това съобщение ще бъде достъпно по-късно чрез свойството **`Message`** на изключението и ще помогне на програмиста, който използва дадения клас, по-лесно да идентифицира проблема.
- Изключенията могат да намалят значително производителността на приложението, понеже всяко хвърлено изключение инстанцира клас (това отнема време), инициализира членовете му (това също отнема време), извършва търсене в стека за подходящ **`catch`** блок (и това отнема време) и накрая след като инстанцията стане неизползваема, тя се унищожава от garbage collector (и това също отнема време). Затова, когато е възможно се препоръчва да се прави проверка дали е възможно дадено действие, а не да се разчита на обработката на възникналото изключение. Прекомерното използване на изключенията се отразява на производителността.
- Някои изключения могат да възникват по всяко време без да ги очакваме (например **`System.OutOfMemoryException`**). Добра практика е да се централизира прихващането на този тип изключения на най-високо ниво например в **`Main()`** метода на програма и да се направи елегантно прекратяване на изпълнението на програмата.
- Изключенията трябва да бъдат хвърляни само при ситуации, които наистина са изключителни и трябва да се обработят. В нормалния ход на програмата (когато не възникват проблеми) не трябва да се хвърлят изключения.

Упражнения

1. Обяснете какво представлява изключенията, кои са силните им страни и кога се препоръчва да се използват.
2. Реализирайте структура от данни **`Student`**, която съдържа информация за студент - име, адрес, курс, специалност, изучавани предмети, оценки и т. н. Добавете подходящи конструктори и свойства за достъп до данните от класа. Сложете проверка за валидност на данните за студента в конструкторите и в свойствата за достъп. При невалидни

данни хвърляйте изключение. Дефинирайте подходящи собствени класове за изключенията, свързани с класа **`Student`**.

Използвана литература

1. Светлин Наков, Обектно-ориентирано програмиране в .NET – http://www.nakov.com/dotnet/lectures/Lecture-3-Object-Oriented-Concepts-v1.0.ppt
2. Jeffrey Richter, Applied Microsoft .NET Framework Programming, Microsoft Press, 2002, ISBN 0735614229
3. Suprotim Agarwal, Getting Started with Exception Handling in C# - http://www.c-sharpcorner.com/Code/2004/July/GettingStartedWithExceptionHandling.asp
4. Steve McConnell, Code Complete, 2nd Edition, Microsoft Press, 2004, ISBN 0735619670
5. MSDN Library – http://msdn.microsoft.com

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 5. Обща система от типове (Common Type System)

Необходими знания

- Базови познания за архитектурата на .NET Framework
- Базови познания за езика C#

Съдържание

- Какво е CTS?
- Йерархията на типовете в .NET
- Стойностни и референтни типове
- Типът **`System.Object`**
- Предефиниране на стандартните методи на **`System.Object`**
- Операторите **`is`** и **`as`**
- Клониране на обекти
- Опаковане и разопаковане на обекти
- Интерфейсите **`IComparable`**, **`IEnumerable`** и **`IEnumerator`**

В тази тема ...

В настоящата тема ще разгледаме общата система от типове в .NET Framework. Ще обясним разликата между стойностни и референтни типове, ще разгледаме основополагащия тип **`System.Object`** и йерархията на типовете, произлизаща от него. Ще се запознаем накратко и с някои операции при работа с типове – преобразуване към друг тип, проверка на тип, клониране, опаковане, разопаковане и др.

Какво е CTS?

CLR поддържа много езици за програмиране. За да се осигури съвместимост на данните между различните езици е разработена общата система от типове (Common Type System – CTS). CTS дефинира поддържаните от CLR типове данни и операциите над тях.

CTS и езиците за програмиране в .NET

Всички .NET езици използват типовете от CTS. За всеки тип в даден .NET език има някакво съответствие в CTS, макар че понякога това съответствие не е директно. Обратното не е вярно – съществуват CTS типове, които не се поддържат от някои .NET езици.

CTS е обектно-ориентирана

По идея всички езици в .NET Framework са обектно-ориентирани. Common Type System също се придържа към идеите на обектно-ориентираното програмиране (ООП) и по тази причина описва освен стандартните типове (числа, символи, низове, структури, масиви) и някои типове данни свързани с ООП (например класове и интерфейси).

CTS описва .NET типовете

Типовете данни в CTS биват най-разнообразни:

- примитивни типове (primitive types – int, float, bool, char, …)
- изброени типове (enums)
- класове (classes)
- структури (structs)
- интерфейси (interfaces)
- делегати (delegates)
- масиви (arrays)
- указатели (pointers)

Всички тези типове повече или по-малко вече са ни познати от езика C#, но всъщност те са част от CTS. Езикът C# и другите .NET езици използват CTS типовете и им съпоставят запазени думи съгласно своя синтаксис. Например типът **`System.Int32`** от CTS съответства на типа **`int`** в C#, а типът **`System.String`** – на типа **`string`**.

Стойностни и референтни типове

В CTS се поддържат две основни категории типове: **стойностни типове** (value types) и **референтни типове** (reference types). Стойностните типове съдържат директно стойността си в стека за изпълнение на програмата, докато референтните типове съдържат строго типизиран указател

(референция) към стойността, която се намира в динамичната памет. По-нататък ще разгледаме подробно разликите между стойностните и референтните типове и особеностите при тяхното използване.

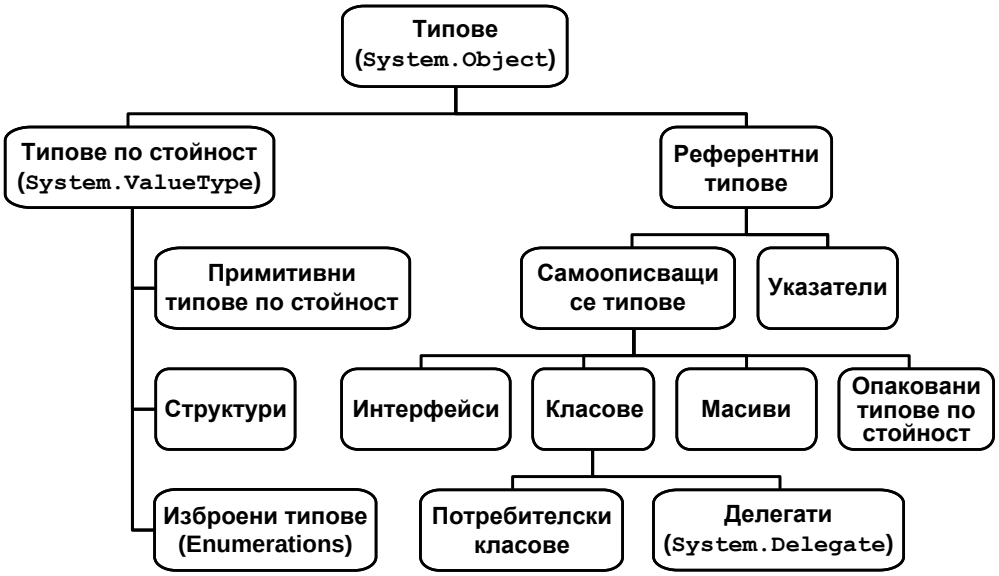
Къде са ми указателите?

По принцип в .NET има класически указатели, но те не се използват масово, както при езиците C и C++. Указателите в .NET се поддържат най-вече заради съвместимост с Win32 платформата и се използват в много специални случаи. В силно типизираните езици като C# и VB.NET за достъп до обекти в динамичната памет се използват т. нар. референции (references), които са строго типизирани указатели, подобни на псевдонимите в C++.

С въвеждането на референтните типове в .NET отпада нуждата от класически указатели. На практика референтните типове са типово-обезопасени указатели, защитени от неправилно преобразуване към друг тип, а сочената от тях динамична памет се управлява автоматично.

Йерархията на типовете

CTS дефинира строга йерархия на типовете данни, които се поддържат в .NET Framework:



В основата на йерархията стои системният тип **`System.Object`**. Той е общ предшественик (базов тип) за всички останали типове в CTS. Неговите преки наследници са стойностните и референтните типове (които ще дискутираме в детайли по-късно в тази тема).

Стойностните типове биват примитивни (**`int`**, **`float`**, **`bool`** и др.), структури (**`struct`** в C#) и изброени типове (**`enum`** в C#).

Референтните типове са всички останали – указателите, класовете, интерфейсите, делегатите, масивите и опакованите стойностни типове.

В предходните теми вече се запознахме с някои от CTS типовете. В тази и в следващите теми ще се запознаем и с останалите (опаковани стойностни типове, масиви, делегати).

Типът System.Object

В CTS всички типове наследяват системния тип **System.Object**. Не правят изключение дори примитивните типове (**int**, **float**, **char**, ...) и масивите. Всеки тип е наследник на **System.Object** и имплементира методите, включени в него. Като резултат значително се улеснява работата с типове, защото променлива от произволен тип може да се присвои на променлива от базовия тип **System.Object** (**object** в C#). Самият **System.Object** е референтен тип.

Стойностни типове (value types)

Стойностни типове (типове по стойност) са повечето примитивни типове (**int**, **float**, **bool**, **char** и др.), структурите (**struct** в C#) и изброените типове (**enum** в C#).

Стойностните типове директно съдържат стойността си и се съхраняват физически в работния стек за изпълнение на програмата. Те не могат да приемат стойност **null**, защото реално не са указатели.

Стойностните типове и паметта

Стойностните типове заемат необходимата им памет в стека в момента на деклариране им и я освобождават в момента на излизане от обхват (при достигане на края на програмния блок, в който са декларирани). Заделянето и освобождаване на памет за стойностен тип реално се извършва чрез единично преместване на указателя на стека и следователно става много бързо.

Горното обяснение е малко опростено. Всъщност ако стойностен тип има за член-данни само стойностни типове, при инстанциране целият тип ще се задели в стека. Ако, обаче, стойностен тип (например структура) съдържа като член-данни референтни типове, стойностите им ще се запишат в динамичната памет.

Стойностните типове наследяват System.ValueType

CLR се грижи всички стойностни типове да наследяват системния тип **System.ValueType**. Всички типове, които не наследяват **ValueType** са референтни типове, т.е. реално са указатели към динамичната памет (адреси в паметта).

Предаване на стойностни типове

При извикване на метод стойностните типове се подават по стойност, т.е. предава се копие от тях. При подготовка на извикването на метод CLR копира подаваните като параметри стойностни типове от оригиналното им местоположение в стека на ново място в стека и подава на извиквания метод направените копия. Ако извикваният метод промени стойността на подадения му по стойност параметър, при връщане от извикването промяната се губи. Това поведение важи, разбира се, само ако параметрите се подават по подразбиране, без да се използват ключовите думи в C# **`ref`** и **`out`**, които ще разгледаме по-нататък в следващите теми.

Референтни типове (reference types)

Референтни типове (типове по референция) са указателите, класовете, интерфейсите, делегатите, масивите и опакованите стойностни типове. Физически референтните типове представляват указател към стойност в динамичната памет, но за CLR те не са обикновени указатели, а специални типово-обезопасени указатели. Това означава, че CLR не допуска на един референтен тип да се присвои стойност от друг референтен тип, който не е съвместим с него (т.е. не е същия тип или негов наследник). В резултат на това в .NET езиците грешките от неправилна работа с типове са силно намалени.

Референтните типове и паметта

Всички референтни типове се съхраняват в **динамичната памет** (т. нар. **managed heap**), която се контролира от системата за почистване на паметта (garbage collector). Динамичната памет е специално място от паметта, заделено от CLR за съхранение на данни, които се създават динамично по време на изпълнението на програмата. Такива данни са инстанциите на всички референтни типове.

Когато инстанция на референтен тип престане да бъде необходима на програмата, тя се унищожава от системата за почистване на паметта (т. нар. garbage collector).

Когато инстанцираме референтен тип с оператора **`new`**, CLR заделя място в динамичната памет, където ще стоят данните и един указател в стека, който съдържа адреса на заделеното място. Веднага след това заделената памет се занулява (освен ако програмистът не инициализира заделената променлива, например чрез извикване на подходящ конструктор).

Ако референтен тип (например клас) съдържа член-данни от стойностен тип, те се съхраняват в динамичната памет. Ако референтен тип съдържа член-данни от референтен тип, в динамичната памет се заделят указатели (референции) за тях, а техните стойности (ако не са **`null`**) също се заделят също в динамичната памет, но като отделни обекти.

Референтните типове и производителността

Понякога се приема, че заделянето на динамична памет е бърза операция, защото в текущата реализация (.NET Framework 1.1) физически се имплементира чрез преместване на един указател. Освобождаването на памет, обаче, е сложна и времеотнемаща операция, която се извършва от време на време от системата за почистване на паметта (garbage collector).

Ако изчислим средното време, необходимо за заделяне и освобождаване на динамична памет, се оказва, че заделянето и освобождаване на стойностните типове е значително по-бързо от референтните типове. Когато производителността е важна за нашата система, трябва да се съобразяваме с особеностите на стойностните и референтните типове и начина, по който те заделят и освобождават памет.

Глобално погледнато, нещата около управлението на динамичната памет в .NET Framework са доста комплексни, но в тази тема няма да се спираме на тях. По-нататък, в темата за управление на паметта и ресурсите, ще им обърнем специално внимание.

Стойностни срещу референтни типове

Стойностните и референтните типове в .NET Framework се различават съществено. Стойностните типове се разполагат в стека за изпълнение на програмата, докато референтните типове са строго типизирани указатели към динамичната памет, където се съдържа самата им стойност.

Следват някои по-съществени разлики между тях:

- Стойностните типове наследяват системния тип **`System.ValueType`**, а референтните наследяват директно **`System.Object`**.
- При създаване на променлива от стойностен тип тя се заделя в стека, а при референтните типове – в динамичната памет.
- При присвояване на стойностни типове се копира самата им стойност, а при референтни типове – само референцията (указателя).
- При предаване на променлива от стойностен тип като параметър на метод, се предава копие на стойността ѝ, а при референтните типове се предава копие на референцията, т.е. самата стойност не се копира. В резултат, ако даден метод променя стойностен входен параметър, промените се губят при излизане от метода, а ако входният параметър е референтен, те се запазват.
- Стойностните типове не могат да приемат стойност **`null`**, защото не са указатели, докато референтните могат.
- Стойностните типове се унищожават при излизане от обхват, докато референтните се унищожават от системата за почистване на паметта (garbage collector) в някой момент, в който се установи, че вече не са необходими за работата на програмата.

- Променливи от стойностен тип могат да се съхраняват в променливи от референтен тип чрез т.нар. "опаковане" (boxing), което ще разгледаме след малко.

Стойностни и референтни типове – пример

В настоящия пример се демонстрира използването на стойностни и референтни типове и се илюстрира разликата между тях:

```
using System;

// SomeClass is reference type
class SomeClass
{
   public int mValue;
}

// SomeStruct is value type
struct SomeStruct
{
   public int mValue;
}

class TestValueAndReferenceTypes
{
   static void Main()
   {
      SomeClass class1 = new SomeClass();
      class1.mValue = 100;
      SomeClass class2 = class1; // Копира се референцията
      class2.mValue = 200; // Променя се и class1.mValue
      Console.WriteLine(class1.mValue); // Отпечатва се 200

      SomeStruct struct1 = new SomeStruct();
      struct1.mValue = 100;
      SomeStruct struct2 = struct1; // Копира се стойността
      struct2.mValue = 200; // Променя се копираната стойност
      Console.WriteLine(struct1.mValue); // Отпечатва се 100
   }
}
```

След като се изпълни примерът, се получава следния резултат:

Как работи примерът?

В началото на примера се създава инстанция на класа **`SomeClass`**, в нея се записва числото 100 и след това тя се присвоява на две променливи. Аналогично се създава инстанция на структурата **`SomeStruct`**, в нея също се записва 100 и след това тя се присвоява на две променливи.

При присвояването на инстанциите на класа, понеже той е референтен тип, се присвоява само референцията и стойността реално не се копира, а остава обща. При присвояването на инстанцията на структурата, понеже тя е стойностен тип, се присвоява самата стойност (нейно копие). Поради тази причина в резултат от изпълнението на програмата на конзолата се отпечатват различни стойности.

По-долу са показани схематично стекът за изпълнение на програмата и динамичната памет в момента преди приключване на програмата. Данните са взети от дебъгера на Visual Studio .NET поради което са много близки до истинското разположение на паметта по време на изпълнение на примерната програма:

Стек за изпълнение на програмата

променлива	адрес	стойност
(начало на стека)	...	...
class1	0x0012F680	0x04A41A44
class2	0x0012F67C	0x04A41A44
struct1	0x0012F678	100
struct2	0x0012F674	200
(свободна стекова памет)	…	…
(край на стека)	0x00000000	…

Динамична памет (managed heap)

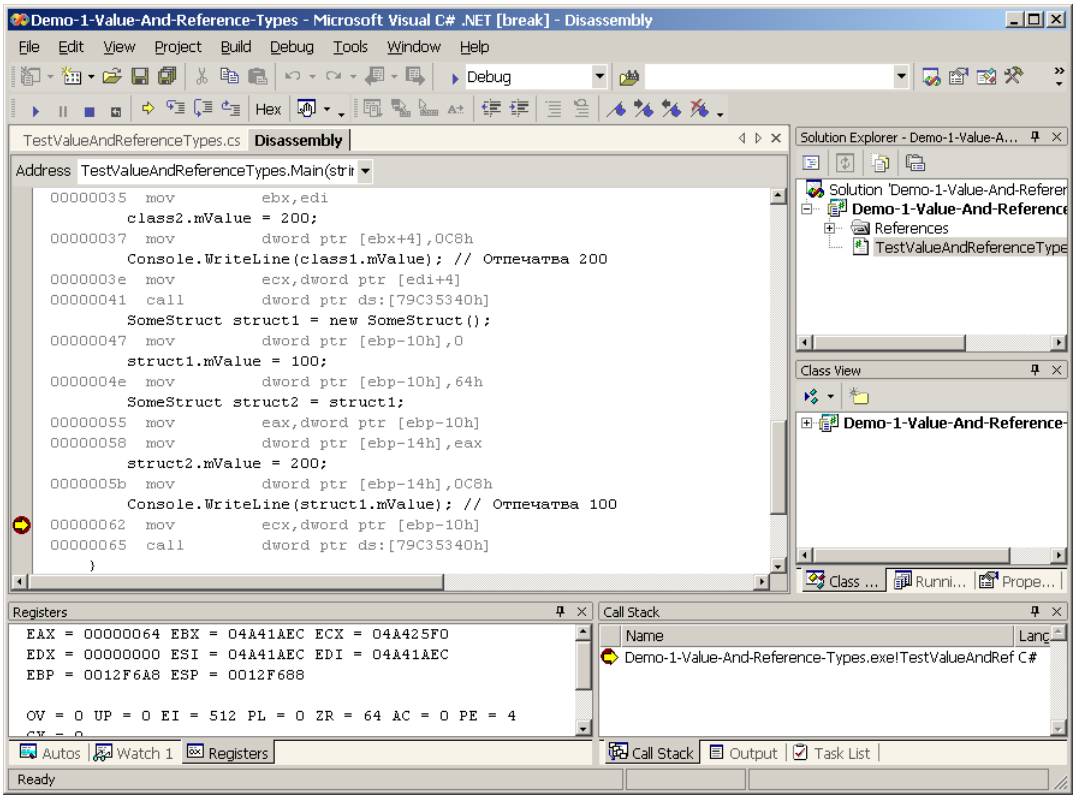
адрес	стойност
…	…
…	…
0x04A41A44	200
…	…
…	…
…	…
…	…

Стекът расте отгоре надолу (от големите адреси към адрес 0), защото програмата е изпълнена върху Intel-съвместима архитектура, при която това поведение е нормално.

Проследяване на примера с VS.NET

За да проследим как се изпълнява горният пример стъпка по стъпка, можем да използваме проекта **`Demo-1-Value-And-Reference-Types`** от демонстрациите:

1. Отваряме с VS.NET проекта **`Demo-1-Value-And-Reference-Types.sln`**.
2. Слагаме точка на прекъсване на последния ред от **`Main()`** метода на основния клас.
3. Стартираме приложението с **`[F5]`**.
4. След като дебъгерът спре в точката на прекъсване, показваме **`Disassembly`** и **`Registers`** прозорците. От менюто на VS.NET избираме **`Debug | Windows | Disassembly`** и **`Debug | Windows | Registers`**. Ето как изглежда VS.NET в този момент:



5. Можем да разгледаме асемблерния код, получен след компилиране на програмата и след превръщането на MSIL кода в чист Win32 код за процесор Intel x86.

   Повечето компилатори за Intel-базирани процесори генерират код, който използва в тялото на методите регистър **`EBP`** като указател към върха на стека. Адресиране от типа на **`dword ptr [ebp-14h]`** най-често реферира стойност в стека – локална променлива или параметър.

   Спомнете си за разликите между класове и структури (референтни и стойностни типове). Стойностните типове съхраняват стойността си

директно в стека. Референтните типове съхраняват в стека само 4-байтов адрес, който указва мястото на променливата в динамичната памет.

Често пъти, с цел оптимизация на производителността, компилаторът вместо някаква област от стека използва регистри за съхранение на локални променливи. В случая в **EBX** се съхранява референцията **class2**, а в **EDI** – референцията **class1**.

6. Да разгледаме асемблерния код, генериран за операцията присвояване **class2=class1**. В него се присвоява на регистър **EBX** стойността на регистър **EDI**, т.е. на референцията **class2** се присвоява референцията **class1**. Обърнете внимание, че се копира референцията, а не самата стойност.
7. Да разгледаме асемблерния код, генериран за операцията присвояване **struct2=struct1**. В него се присвоява на регистър **EAX** стойността от стека, съответстваща на **struct1** и след това стойността от **EAX** се записва обратно в стека, в променливата **struct2**. На практика се копира самата стойност на структурата, като се използва за работна променлива регистърът **EAX**.

Защита от неинициализирани променливи

Когато декларираме променлива в кода, C# компилаторът ни задължава да й зададем стойност преди първото й използване. Ако се опитаме да използване неинициализирана променлива (независимо дали е от стойностен или референтен тип), C# компилаторът дава грешка и отказва да компилира кода. Ето един пример:

```
int someVariable;
Console.WriteLine(someVariable);
```

При опит за компилация ще възникне грешката "Use of unassigned local variable someVariable".

Автоматична инициализация на променливите

При създаване на обект от даден тип с оператора **new** CLR автоматично инициализира декларираната променлива с неутрална (нулева) стойност. Ето един пример:

```
int i = new int();
Console.WriteLine(i);
```

Горният код се компилира успешно и отпечатва като резултат **0**. Това се дължи на автоматичната инициализация, която операторът **new** извършва.

Когато заделяме структура или клас, се изпълнява и съответният конструктор и всички член-променливи на новия обект се инициализират с

нулеви стойности. Това предпазва разработчиците от проблеми свързани с неинициализирани член-данни, които могат да бъдат много досадни, защото се проявяват само от време на време.

Ако само дефинираме променлива, без да създадем инстанция за нея с оператора **new**, ще получим грешка по време на компилация, защото променливата ще остане неинициализирана. Ето пример:

```
int i;
Console.WriteLine(i); // Use of unassigned local variable 'i'
```

Типът System.Object

Типът **System.Object** е базов за всички типове в .NET Framework. Както референтните, така и стойностните типове произлизат от **System.Object** или от негов наследник. Това улеснява програмиста и в много ситуации му спестява писане на излишен код.

В .NET Framework можем да напишем следния код:

```
string s = 5.ToString();
```

Този код извиква виртуалния метод **ToString()** от класа **System.Object**. Това е възможно, защото числото **5** е инстанция на типа **System.Int32**, който е наследник на **System.Object**.

Понеже всички типове са съвместими със **System.Object** (**object** в C#), защото са негови наследници, можем на инстанция на **System.Object** да присвояваме както референтни, така и стойностни типове:

```
object obj = 5;
object obj2 = new SomeClass();
```

Забележка: Ако не е указано друго, в C# целите числа по подразбиране са инстанции на типа **System.Int32**.

Защо стойностните типове наследяват референтния тип System.Object?

Ако си спомним, че **System.Object** е референтен тип, изглежда малко странно че стойностните типове също го наследяват. Сякаш има някакво противоречие: Как така стойностните типове, които не са указатели, произлизат от тип, който е указател?

Всъщност противоречие няма, защото архитектите на .NET Framework по изкуствен начин са направили съвместими всички стойностни типове със **System.Object**. За удобство в CLR всички стойностни типове могат да се преобразуват към референтни чрез операцията "**опаковане**". Опаковането и обратната му операция "**разопаковане**" преобразуват стойностни

типове в опаковани стойностни типове и обратното. При опаковане стойностните типове се копират в динамичната памет и се получава указател (референция) към тях. При разопаковане стойността от динамичната памет, сочена от съответната референция, се копира в стека.

По късно в настоящата тема ще дискутираме в детайли опаковането и разопаковането на стойностни типове.

Потребителските типове скрито наследяват System.Object

При дефиниране на какъвто и да е тип, скрито от нас се наследява **System.Object**. Например структурата:

```
struct Point
{
  int x, y;
}
```

е наследник на **System.Object**, макар това да не се вижда непосредствено от декларацията й.

Методите на System.Object

Като базов тип за всички .NET типове **System.Object** дефинира обща за всички тях функционалност. Тази функционалност се реализира в няколко метода, някои от които са виртуални и могат да бъдат припокрити:

- **bool Equals(object)** – виртуален метод, който сравнява текущия обект с друг обект. Методът има и статична версия **Equals(object, object)**, която сравнява два обекта, подадени като параметри. Обектите се сравняват не по адрес, а по съдържание. Методът често пъти се припокрива, за да се даде възможност за сравнение на потребителски обекти.
- **string ToString()** – виртуален метод, който представя обекта във вид на символен низ. Имплементацията по подразбиране на **ToString()** отпечатва името самия тип.
- **int GetHashCode()** – виртуален метод за изчисляване на хеш-код. Използва се при реализацията на някои структури от данни, например хеш-таблици. По-нататък, в темата за масиви и колекции, ще разгледаме този метод по-детайлно.
- **Finalize()** – виртуален метод за имплементиране на почистващи операции при унищожаване на обект. В C# не може да се дефинира директно, а се имплементира чрез деструктора на типа. Ще разгледаме подробности за т. нар. "финализация на обекти" в темата за управление на паметта и ресурсите.
- **Type GetType()** – връща метаданни за типа на обекта във вид на инстанция на **System.Type**. Имплементиран е вътрешно от CLR.

- **object MemberwiseClone()** – копира двоичното представяне на обекта в нов обект, т. е. извършва плитко копиране. При референтни типове създава нова референция към същия обект. При стойностни типове копира стойността на подадения обект.
- **bool ReferenceEquals()** – сравнява два обекта по референция. При референтни типове се сравнява дали обектите сочат на едно и също място в динамичната памет. При стойностни типове връща **false**.

Предефиниране на сравнението на типове

Когато дефинираме собствен тип, често пъти се налага да се имплементира функционалност за сравнение на негови инстанции. В .NET Framework се препоръчва такава функционалност да се реализира чрез имплементиране на предвидените за целта методи в **System.Object**.

Препоръчва се методите **Equals(object)**, **operator ==**, **operator !=** и **GetHashCode()** да се имплементират заедно в комплект. Тази практика спестява някои доста досадни проблеми. Например ако **Equals(object)** е имплементиран, а операторът **==** не е имплементиран, потребителите на типа могат да се подведат и да извършват некоректно сравнение с **==**, което по подразбиране връща резултата от метода **ReferenceEquals()**.

Предефиниране на сравнението – пример

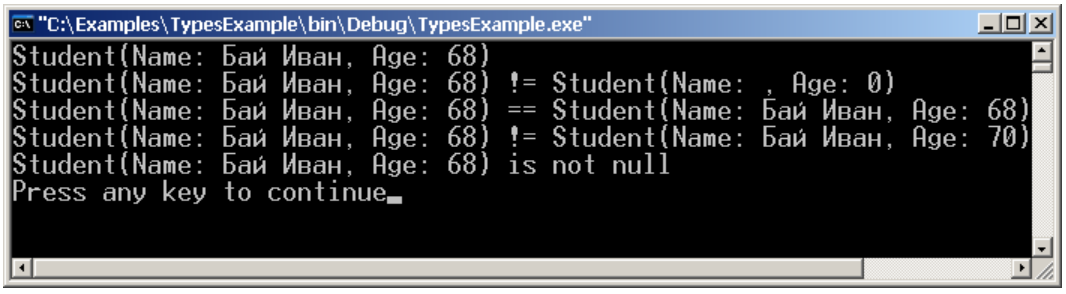
В настоящия пример се дефинира клас **Student**, който съдържа 2 информационни полета (име и възраст), след което се дефинират методите за сравнение на студенти. Счита се, че два студента са един и същ, ако имат еднакви имена и възраст. Предефинират се виртуалните методи **Equals(object)**, **operator ==**, **operator !=**, **GetHashCode()** и **ToString()** от **System.Object**. С цел илюстриране как се използва предефинираното сравнение в края на примера се създават няколко инстанции на **Student** и се сравняват една с друга.

```
using System;

public class Student
{
   public string mName;
   public int mAge;

   public override bool Equals(object aObject)
   {
      // If the cast is invalid, the result will be null
      Student student = aObject as Student;

      // Check if we have valid not null Student object
      if (student == null)
      {
         return false;
```

```
    }

    // Compare the reference type member fields
    if (! Object.Equals(this.mName, student.mName))
    {
      return false;
    }

    // Compare the value type member fields
    if (this.mAge != student.mAge)
    {
      return false;
    }

    return true;
  }

  public static bool operator == (Student aStudent1,
    Student aStudent2)
  {
    return Student.Equals(aStudent1, aStudent2);
  }

  public static bool operator != (Student aStudent1,
    Student aStudent2)
  {
    return ! (Student.Equals(aStudent1, aStudent2));
  }

  public override int GetHashCode()
  {
    // Return the hash code of the mName field
    return mName.GetHashCode();
  }

  public override string ToString()
  {
    return String.Format(
      "Student(Name: {0}, Age: {1})", mName, mAge);
  }

  static void Main()
  {
    Student st1 = new Student();
    st1.mName = "Бай Иван";
    st1.mAge = 68;
    Console.WriteLine(st1); // Student.ToString() is called

    Student st2 = new Student();
    if (st1 != st2)  // it is true
```

```
        {
            Console.WriteLine("{0} != {1}", st1, st2);
        }

        st2.mName = "Бай Иван";
        st2.mAge = 68;
        if (st1 == st2) // it is true
        {
            Console.WriteLine("{0} == {1}", st1, st2);
        }

        st2.mAge = 70;
        if (st1 != st2)  // it is true
        {
            Console.WriteLine("{0} != {1}", st1, st2);
        }

        if (st1 != null) // it is true
        {
            Console.WriteLine("{0} is not null", st1);
        }
    }
}
```

След като се изпълни примерът, се получава следния резултат:



Как работи примерът?

Методът **`Equals(object)`** е реализиран на няколко стъпки. Първо се проверява дали е подаден обект от тип **`Student`**, който не е **`null`**. Това е необходимо условие, за да е възможно равенството на подадения студент с текущия студент. След това се сравняват имената на студентите и ако съвпаднат се сравняват и годините им. Истина се връща, само ако и двете сравнения установят равенство.

Операторите **`==`** и **`!=`** се имплементират чрез извикване на **`Equals(object)`**.

Методът **`GetHashCode()`** връща хеш-кода на името на студента, което ще върши работа в повечето случаи. По-подробно на този метод ще се спрем в темата "Масиви и колекции".

Методът **ToString()** връща символен низ, съдържащ името и възрастта на студента в лесно четим формат.

В главната програма (**Main()** метода) се извършват серия сравнения, които демонстрират правилната работа на имплементираните методи.

Оператори за работа с типове в C#

В C# има няколко служебни оператора за работа с типове – **is** и **as** и **typeof**.

Оператор is

Операторът **is** проверява дали зададеният обект е инстанция на даден тип. Пример:

```
int value = 5;
if (value is System.Object) // it is true
{
   Console.WriteLine("{0} is instance of System.Object.", value);
}
```

Оператор as

Операторът **as** преобразува даден референтен тип в друг, като при неуспех не предизвиква изключение, а връща стойност **null**. При стандартно преобразуване на типове, ако има несъвместимост на обекта с резултатния тип, се получава изключение. Например:

```
int i = 5;
object obj = i;
string str = (string) obj; // System.InvalidCastException
```

Операторът **as** преобразува типове, без да предизвиква изключение:

```
int i = 5;
object obj = i;
string str = obj as string; // str == null
```

Оператор typeof

Операторът **typeof** извлича отражението на даден тип във вид на инстанция на **System.Type**. Пример:

```
Type intType = typeof(int);
```

В темата "Отражение на типовете" ще обърнем повече внимание на типа **System.Type** и на оператора **typeof**.

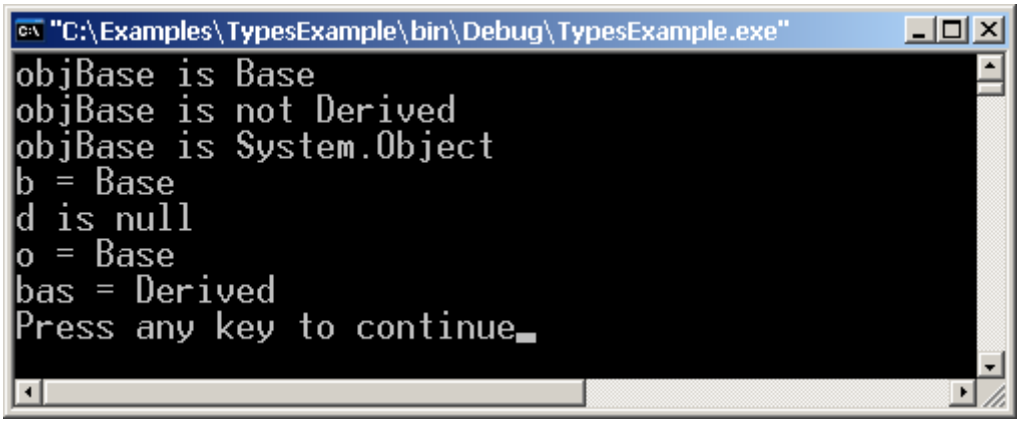
Оператори is и as – пример

В следващия пример се илюстрира използването на операторите **is** и **as**:

```
using System;

class Base
{

}

class Derived : Base
{

}

class TestOperatorsIsAndAs
{
   static void Main()
   {
      Object objBase = new Base();
      if (objBase is Base)
      {
         Console.WriteLine("objBase is Base");
      }
      // Result: objBase is Base

      if (! (objBase is Derived))
      {
         Console.WriteLine("objBase is not Derived");
      }
      // Result: objBase is not Derived

      if (objBase is System.Object)
      {
         Console.WriteLine("objBase is System.Object");
      }
      // Result: objBase is System.Object

      Base b = objBase as Base;
      Console.WriteLine("b = {0}", b);
      // Result: b = Base

      Derived d = objBase as Derived;
      if (d == null)
      {
         Console.WriteLine("d is null");
      }
      // Result: d is null

      Object o = objBase as Object;
```

```
        Console.WriteLine("o = {0}", o);
        // Result: o = Base

        Derived der = new Derived();
        Base bas = der as Base;
        Console.WriteLine("bas = {0}", bas);
        // Result: bas = Derived
    }
}
```

Примерът декларира два класа – **Base** и негов наследник **Derived**, след което създава няколко инстанции от тези класове и ги преобразува една към друга. Работата на операторите **is** и **as** се илюстрира чрез няколко преобразувания и проверки на типовете. Резултатът от изпълнението на примерната програма е следния:



Клониране на обекти

Клонирането (копирането) на обекти е операция, която създава идентично копие на даден обект. При клонирането се създават копия на всички информационни полета (член-променливи) на типа. Съществуват 2 типа клониране – плитко и дълбоко.

Плитко клониране

При плитко клониране всички стойностни типове се копират, а всички референции се дублицират (копират се адресите). На практика се създава копие на обекта, което може да има общи (споделени) части с оригиналния обект (това са всички полета на оригиналния обект, които са от референтен тип). Плитко клониране се извършва от метода **MemberwiseClone()** на типа **System.Object**.

Дълбоко клониране

При дълбоко (пълно) клониране се правят копия на всички полета на оригиналния обект и се създава съвсем нов обект, който е идентичен с оригиналния, но не съдържа споделени с него общи данни. На практика се дублицират рекурсивно в дълбочина всички полета на оригиналния обект и съответно техните полета.

Плитки срещу дълбоки копия

В програмирането използването на плитки копия на обектите често води до проблеми и затова не е препоръчвана практика. Когато трябва да се клонира даден обект, обикновено е необходимо да се създаде негово пълно копие, а не само нова референция, сочеща към оригиналния обект.

В някои редки случаи, от съображения за производителност и пестене на ресурси, се налага да се ползват плитки или частични копия на обектите. Ако се прилагат такива техники, това трябва да се прави много внимателно, за да не се получават странни проблеми, като синдромът "ама това вчера работеше".

Интерфейсът ICloneable

В .NET Framework под клониране се подразбира "дълбоко клониране". Всички типове, които позволяват клониране, трябва да имплементират интерфейса **`System.ICloneable`**.

`ICloneable` дефинира метод **`Clone()`** който връща идентично копие на обекта. **`Clone()`** методът трябва да връща дълбоко копие на оригиналния обект. Ако даден обект съдържа като член-данни други обекти, тези обекти трябва също да имплементират **`ICloneable`** и да бъдат клонирани посредством **`Clone()`** метода им. Ако това не бъде изпълнено, има вероятност клонирането да не работи правилно и да се получат споделени данни между оригиналния обект и копието.

Клониране на обекти в .NET Framework

Клонирането като цяло е проблем, при който често възникват грешки, но за щастие рядко се налага да бъде имплементирано ръчно.

Голяма част от често използваните стандартни типове в .NET Framework имат имплементация на **`ICloneable`** – масивите, колекциите, символните низове и др. Примитивните стойностни типове (**`int`**, **`float`**, **`double`**, **`byte`**, **`char`** и т. н.) могат да бъдат клонирани чрез просто присвояване, защото не съдържат вложени членове от референтен тип. При тях на практика всяко клониране е дълбоко.

Имплементиране на ICloneable – пример

В следващия пример ще илюстрираме как може да се клонира нетривиална структура от данни, а именно динамично реализиран свързан списък. При него всеки елемент съдържа някаква стойност и референция към следващ елемент. Последният елемент съдържа за следващ елемент стойност **null**.

При клонирането на свързан списък трябва да се клонират всичките му елементи и връзките между тях. В резултат трябва да се построи нов списък, който съдържа елементите от първия в реда, в който са били в него. На практика клонирането на свързан списък се свежда до обхождането му и построяването на копие на всеки негов елемент и на всяка връзка между два елемента. Следва примерна реализация:

```
using System;
using System.Text;

class LinkedList : ICloneable
{
   public string mValue;
   protected LinkedList mNextNode;

   public LinkedList(string aValue, LinkedList aNextNode)
   {
      mValue = aValue;
      mNextNode = aNextNode;
   }

   public LinkedList(string aValue) : this(aValue, null)
   {
   }

   // Explicit implementation of ICloneable.Clone()
   object ICloneable.Clone()
   {
      return this.Clone();
   }

   // This method is not ICloneable.Clone()
   public LinkedList Clone()
   {
      // Clone the first element
      LinkedList original = this;
      string value = original.mValue;
      LinkedList result = new LinkedList(value);
      LinkedList copy = result;
      original = original.mNextNode;

      // Clone the rest of the list
```

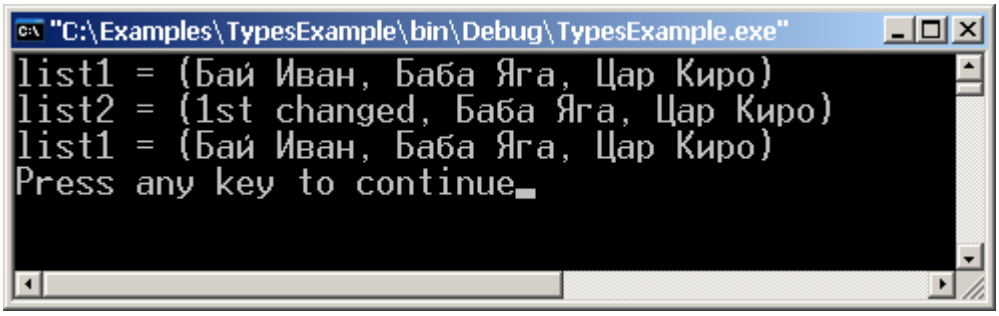
```
      while (original != null)
      {
         value = original.mValue;
         copy.mNextNode = new LinkedList(value);
         original = original.mNextNode;
         copy = copy.mNextNode;
      }

      return result;
   }

   public override string ToString()
   {
      LinkedList currentNode = this;
      StringBuilder sb = new StringBuilder("(");
      while (currentNode != null)
      {
         sb.Append(currentNode.mValue);
         currentNode = currentNode.mNextNode;
         if (currentNode != null)
         {
            sb.Append(", ");
         }
      }
      sb.Append(")");

      return sb.ToString();
   }
}

class TestClone
{
   static void Main()
   {
      LinkedList list1 =
         new LinkedList("Бай Иван",
         new LinkedList("Баба Яга",
         new LinkedList("Цар Киро")));

      Console.WriteLine("list1 = {0}", list1);
      // Result: list1 = (Бай Иван, Баба Яга, Цар Киро)

      LinkedList list2 = list1.Clone();
      list2.mValue = "1st changed";

      Console.WriteLine("list2 = {0}", list2);
      // Result: list2 = (1st changed, Баба Яга, Цар Киро)

      Console.WriteLine("list1 = {0}", list1);
      // Result: list1 = (Бай Иван, Баба Яга, Цар Киро)
```

```
    }
}
```

В примерната реализация е дефиниран свързан списък от символни низове. Методът **ICloneable.Clone()** е реализиран експлицитно (явно). Допълнително за удобство е дефиниран метод **Clone()**. Разликата между двата метода е във връщания тип. Имплементацията на интерфейса **ICloneable** (методът **ICloneable.Clone()**) връща **object** и ако се използва, трябва да се извършва преобразуване. Методът **Clone()** връща директно правилния тип и ни спестява преобразуването.

Методът **ToString()** използва специалния клас **StringBuilder** за по-ефективно сглобяване на резултатния низ. Класът **StringBuilder** и причините за използването му ще бъдат разгледани подробно в темата за работа със символни низове.

Главната програма създава списък **list1**, съдържащ 3 елемента, и го отпечатва. След това го клонира в променливата **list2** и променя първия му елемент. Тъй като оригиналният списък и неговото копие не съдържат споделени данни, оригиналният списък не се променя и това ясно личи от изведения резултат:



Опаковане (boxing) и разопаковане (unboxing) на стойностни типове

Вече обяснихме, че стойностните типове се съхраняват в стека на приложението и не могат да приемат стойност **null**, докато референтните типове съдържат указател (референция) към стойност в динамичната памет и могат да бъдат **null**.

Понякога се налага на референтен тип да се присвои обект от стойностен тип. Например може да се наложи в **System.Object** инстанция да се запише **System.Int32** стойност. CLR позволява това благодарение на т. нар. "**опаковане**" на стойностните типове (**boxing**).

В .NET Framework стойностните типове могат да се използват без преобразуване навсякъде, където се изискват референтни типове. При нужда CLR опакова и разопакова стойностните типове автоматично. Това

спестява дефинирането на обвиващи (wrapper) класове за примитивните типове, структурите и изброените типове, но разбира се, може да доведе и до някои проблеми, които ще дискутираме по-късно.

Опаковане (boxing) на стойностни типове

Опаковането (boxing) е действие, което преобразува стойностен тип в референция към опакована стойност. То се извършва, когато е необходимо да се преобразува стойностен тип към референтен тип, например при преобразуване на **Int32** към **Object**:

```
int i = 5;
object obj = i; // i се опакова
```

Всяка инстанция на стойностен тип може да бъде опакована чрез просто преобразуване до **System.Object**. Ако един тип е вече опакован, той не може да бъде опакован втори път и при преобразуване към **System.Object** си остава опакован само веднъж.

CLR извършва опаковането по следния начин:

1. Заделя динамична памет за създаване на копие на обекта от стойностния тип.
2. Копира съдържанието на стойностната променливата от стека в заделената динамична памет.
3. Връща референция към създадения обект в динамичната памет.

При опаковането в динамичната памет се записва информация, че референцията съдържа опакован обект и се запазва името на оригиналния стойностен тип.

Разопаковане (unboxing) на опаковани типове

Разопаковането (**unboxing**) е процесът на извличане на опакована стойност от динамичната памет. Разопаковане се извършва при преобразуване на опакована стойност обратно към инстанция на стойностен тип, например при преобразуване на **Object** към **Int32**:

```
object obj = 5; // 5 се опакова
int value = (int) obj; // стойността на obj се разопакова
```

CLR извършва разопаковането по следния начин:

1. Ако референцията е **null** се предизвиква **NullReferenceException**.
2. Ако референцията не сочи към валидна опакована стойност от съответния тип, се предизвиква изключение **InvalidCastException**.
3. Ако референцията е валидна опакована стойност от правилния тип, стойността се извлича от динамичната памет и се записва в стека.

За разлика от опаковането, разопаковането невинаги е успешна операция (и това трябва да се съобразява, когато се работи с опаковани стойности).

Особености при опаковането и разопаковането

При използване на автоматично опаковане и разопаковане на стойности трябва да се имат предвид някои особености:

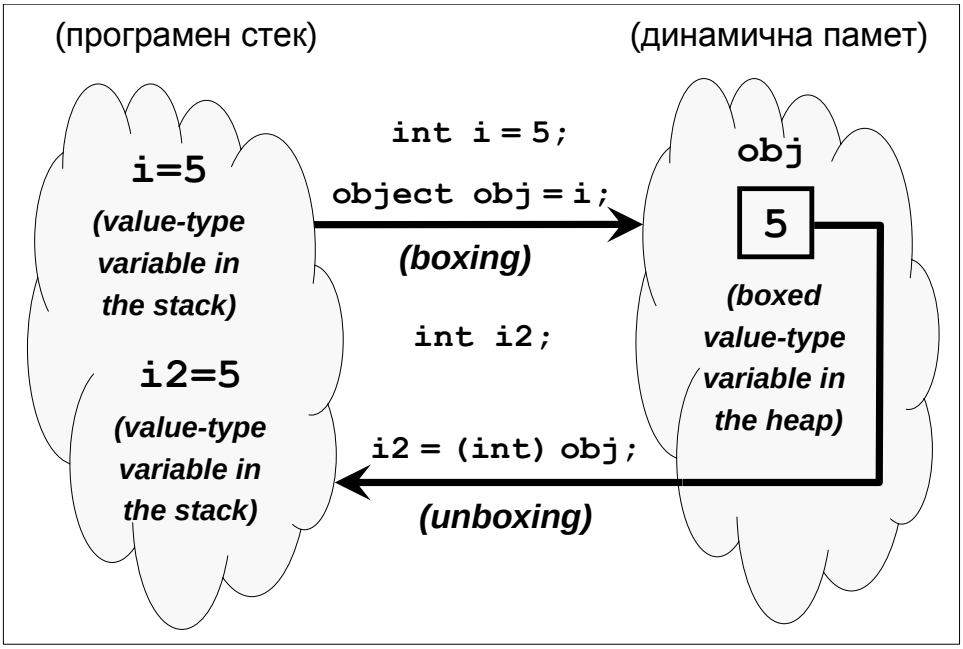
- Опаковането и разопаковането намаляват производителността. За оптимална производителност трябва да се намали броят на опакованите и разопакованите обекти.
- Опакованите типове са копия на оригиналните стойности, поради което, ако променяме оригиналния неопакован тип, опакованото копие не се променя.

Как работят опаковането и разопаковането?

Нека имаме следния код:

```
int i = 5;
object obj = i; // boxing

int i2;
i2 = (int) obj; // unboxing
```

На картинката по-долу схематично е показано как работят опаковането и разопаковането на стойностни типове в .NET Framework:



При опаковане стойността от стека се копира в динамичната памет, а при разопаковане стойността от динамичната памет се копира в обратно в стека.

Опакованите стойности се държат като останалите референтни типове – разполагат се в динамичната памет, унищожават се от garbage collector, когато не са необходими на програмата, и при подаване като параметър при извикване на метод се пренасят по адрес.

Пример за опаковане и разопаковане

В следващия пример се илюстрира опаковането и разопаковането на стойностни типове, като се обръща внимание на някои особености при тези операции:

```
using System;

class TestBoxingUnboxing
{
   static void Main()
   {
      int value1 = 1;
      object obj = value1;  // извършва се опаковане

      value1 = 12345; // променя се само стойността в стека

      int value2 = (int)obj;  // извършва се разопаковане
      Console.WriteLine(value2); // отпечатва се 1

      long value3 = (long) (int) obj; // разопаковане

      long value4 = (long) obj; // InvalidCastException
   }
}
```

От примера се вижда, че разопаковане на **Int32** стойност не може да се извърши чрез директно преобразуване към **Int64**. Необходимо е първо да се извлече **Int32** стойността от опакования обект и след това да се извърши преобразуване до **Int64**.

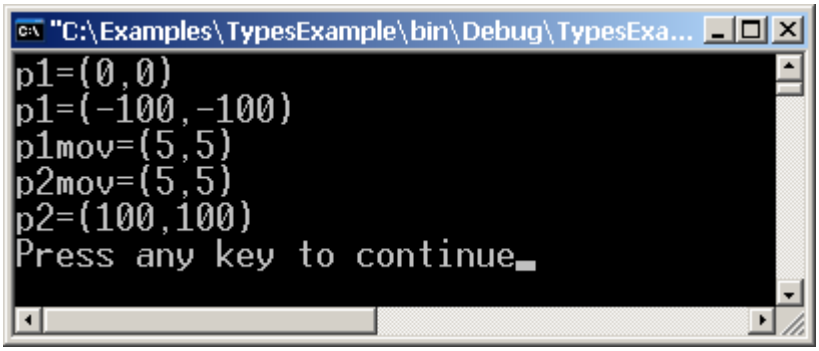
Аномалии при опаковане и разопаковане

При работа с опаковани обекти трябва да се внимава, защото ако не бъдат съобразени някои особености, може да се наблюдава странно поведение на програмата. Ето един такъв пример:

```
using System;

interface IMovable
{
```

```
   void Move(int aX, int aY);
}

/// <summary>
/// Много лоша практика! Структурите не бива
/// да съдържат логика, а само данни!
/// </summary>
struct Point : IMovable
{
   public int mX, mY;

   public void Move(int aX, int aY)
   {
      mX += aX;
      mY += aY;
   }

   public override string ToString()
   {
      return String.Format("({0},{1})", mX, mY);
   }
}

class TestPoint
{
   static void Main()
   {
      Point p1 = new Point();
      Console.WriteLine("p1={0}", p1); // p1=(0,0)

      IMovable p1mov = (IMovable) p1; // p1 се опакова
      IMovable p2mov =      // p1mov не се опакова втори
         (IMovable) p1mov; // път, защото е вече опакован
      Point p2 = (Point) p2mov; // p2mov се разопакова

      p1.Move(-100,-100);
      p2mov.Move(5,5);
      p2.Move(100,100);

      Console.WriteLine("p1={0}", p1); // p1=(-100,-100)
      Console.WriteLine("p1mov={0}", p1mov); // p1mov=(5,5)
      Console.WriteLine("p2mov={0}", p2mov); // p2mov=(5,5)
      Console.WriteLine("p2={0}", p2); // p2=(100,100)
   }
}
```

Резултатът от изпълнение на примера е следният:



Основната причина за този резултат е фактът, че при преобразуване към интерфейс структурите се опаковат и съответно се създава копие на данните, намиращи се в тях. Опаковането е съвсем в реда на нещата, като се има предвид, че структурите са стойностни типове, а интерфейсите са референтни типове.

Препоръчва се, когато се използват структури в C#, те да съдържат само данни. Лоша практика е в структура да се дефинират методи с логика, както и структура да имплементира интерфейс.

Как работи примерът?

Да разгледаме как работи примерът. Ако съобразим разположението на стойностните и референтните променливи в паметта, можем да си обясним какво се случва:

Стек за изпълнение на програмата		
променлива	адрес	стойност
(начало на стека)	...	...
p1	0x0012F6A0	(-100, -100)
p1mov	0x0012F69C	0x04A438B4
p2mov	0x0012F698	0x04A438B4
p2	0x0012F690	(100, 100)
(свободна стекова памет)	...	...
(край на стека)	0x00000000	...

Динамична памет (managed heap)	
адрес	стойност
...	...
...	...
...	...
0x04A438B4	(5, 5)
...	...
...	...
...	...
...	...

Променливите **p1** и **p2** са от стойностен тип и се разполагат директно в стека (и заемат по 8 байта от него).

Променливите **p1mov** и **p2mov** са от референтен тип и се разполагат в динамичната памет. В стека за тях се пазят по 4 байта, които съдържат адреса на стойността им.

С помощта на дебъгера на VS.NET можем да проследим точното разположение и стойностите на тези променливи. В горната таблица е показано състоянието им точно преди завършване на програмата.

Напомняме, че при Intel архитектурата стекът расте надолу и свършва на адрес 0x00000000.

Интерфейсът IComparable

Често пъти освен за равенство е необходимо обектите да се сравняват спрямо някаква подредба (например лексикографска за низове или по големина за числови типове). В .NET Framework типовете, които могат да бъдат сравнявани един с друг, трябва да имплементират интерфейса **System.IComparable**.

Интерфейсът дефинира един-единствен метод – **CompareTo(object)**. Този метод трябва да реализира сравняването и да връща:

- **число < 0** – ако подаденият обект е по-голям от **this** инстанцията
- **0** – ако подаденият обект е равен на **this** инстанцията
- **число > 0** – ако подаденият обект е по-малък от **this** инстанцията

IComparable се използва от .NET Framework при сортиране на масиви и колекции и при някои други операции, изискващи сравнение по големина.

Системни имплементации на IComparable

IComparable е имплементиран от много системни .NET типове, като например от примитивните стойностни типове **System.Char**, **System.Int32**, **System.Single**, **System.Double**, от символните низове (**System.String**) и от изброените типове (**System.Enum**). Това улеснява разработчиците при всекидневната им работа и често пъти им спестява излишни усилия.

Имплементиране на IComparable – пример

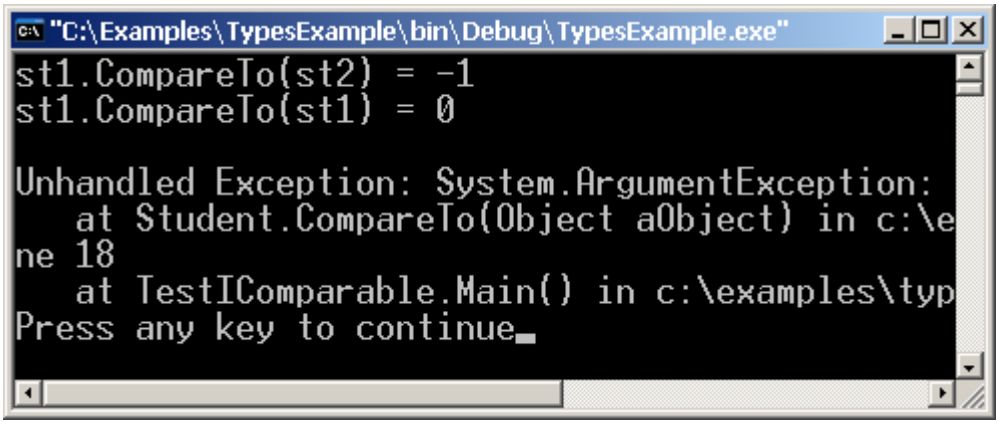
В следващия пример е илюстрирано как можем да имплементираме **IComparable** за потребителски дефинирани типове:

```
using System;

class Student : IComparable
{
```

```
   private string mFirstName;
   private string mLastName;

   public Student(string aFirstName, string aLastName)
   {
      mFirstName = aFirstName;
      mLastName = aLastName;
   }

   public int CompareTo(object aObject)
   {
      if (! (aObject is Student))
      {
         throw new ArgumentException(
            "The object is not Student.");
      }

      Student student = (Student) aObject;
      int firstNameCompareResult =
         String.Compare(this.mFirstName, student.mFirstName);
      if (firstNameCompareResult != 0)
      {
         return firstNameCompareResult;
      }
      else
      {
         int lastNameCompareResult =
            String.Compare(this.mLastName, student.mLastName);
         return lastNameCompareResult;
      }
   }
}

class TestIComparable
{
   static void Main()
   {
      Student st1 = new Student("Бате", "Киро");
      Student st2 = new Student("Кака", "Мара");

      Console.WriteLine(
         "st1.CompareTo(st2) = {0}", st1.CompareTo(st2));
      // Result: -1

      Console.WriteLine(
         "st1.CompareTo(st1) = {0}", st1.CompareTo(st1));
      // Result: 0

      Console.WriteLine(
         "st1.CompareTo(42) = {0}", st1.CompareTo(42));
```

```
        // Result: System.ArgumentException
    }
}
```

В примера се дефинира клас **Student**, който съдържа две информационни полета – име и фамилия. Имплементацията на **CompareTo()** извършва лексикографско сравнение на студенти – първо по име, а след това по фамилия при еднакви имена. Ето как изглежда изходът от примера:



Интерфейсите IEnumerable и IEnumerator

В програмирането се срещат типове, които съдържат много на брой инстанции на други типове. Такива типове се наричат **контейнери** или още **колекции**. Колекции например са масивите, защото съдържат много на брой еднакви елементи.

Често пъти се налага да се обходят всички елементи на дадена колекция. За да става това по стандартен начин, в .NET Framework са дефинирани интерфейсите **IEnumerable** и **IEnumerator**.

Интерфейсът IEnumerable

Интерфейсът **System.IEnumerable** се имплементира от колекции и други типове, които поддържат операцията "обхождане на елементите им в някакъв ред". Този интерфейс дефинира само един метод – методът **GetEnumerator()**. Той връща итератор (инстанция на **IEnumerator**) за обхождане на елементите на дадения обект.

Обектите, поддържащи **IEnumerable** интерфейса, могат да се използват от конструкцията **foreach** в C# за обхождане на всичките им елементи.

Интерфейсът **IEnumerable** е реализиран от много системни .NET типове, като **System.Array**, **System.String**, **ArrayList**, **Hashtable**, **Stack**, **Queue**, **SortedList** и др. с цел да се улесни работата с тях.

Интерфейсът IEnumerator

Интерфейсът **System.IEnumerator** имплементира обхождане на всички елементи на колекции и други типове. Той реализира прост итератор чрез следните методи и свойства:

- Свойство **Current** – връща текущия елемент.
- Метод **bool MoveNext()** – преминава към следващия елемент и връща **true**, ако той е валиден.
- Метод **Reset()** – премества итератора непосредствено преди първия елемент (установява го в начално състояние).

Имплементиране на IEnumerable и IEnumerator

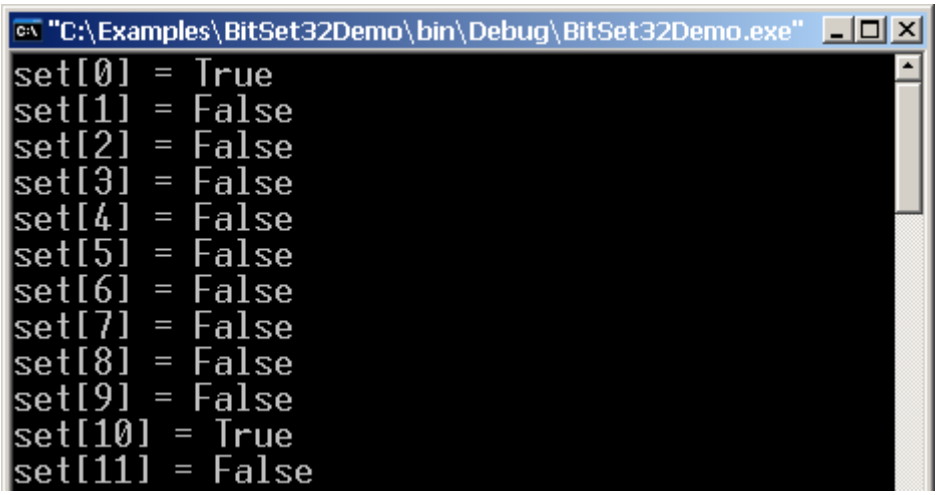
Следващият пример илюстрира как могат да бъдат имплементирани интерфейсите **IEnumerable** и **IEnumerator**, след което да бъдат използвани във **foreach** конструкция в C#:

```
using System;
using System.Collections;

class BitSet32 : IEnumerable
{
  private uint mBits = 0;

  public void Set(int aIndex, bool aValue)
  {
    if (aIndex<0 || aIndex>31)
    {
      throw new ArgumentException("Invalid index!");
    }

    uint bitMask = (uint) 1 << aIndex;

    // Set bit aIndex to 0
    mBits = mBits & (~bitMask);

    if (aValue)
    {
      // Set bit aIndex to 1
      mBits = mBits | bitMask;
    }
  }

  public bool Get(int aIndex)
  {
    if (aIndex<0 || aIndex>31)
    {
      throw new ArgumentException("Invalid index!");
    }
```

```
      uint bitMask = (uint) 1 << aIndex;
      bool value = ((mBits & bitMask) != 0);
      return value;
   }

   public IEnumerator GetEnumerator()
   {
      return new BitSet32Enumerator(this);
   }

   class BitSet32Enumerator : IEnumerator
   {
      private BitSet32 mBitSet32;
      private int mCurrentIndex = -1;

      public BitSet32Enumerator(BitSet32 aBitSet32)
      {
         mBitSet32 = aBitSet32;
      }

      public bool MoveNext()
      {
         mCurrentIndex++;
         bool valid = (mCurrentIndex < 32);
         return valid;
      }

      public void Reset()
      {
         mCurrentIndex = -1;
      }

      public object Current
      {
         get
         {
            return mBitSet32.Get(mCurrentIndex);
         }
      }
   }
}

class TestBitSet32
{
   static void Main()
   {
      BitSet32 set = new BitSet32();
      set.Set(0, true);
      set.Set(31, true);
```

```
      set.Set(5, true);
      set.Set(5, false);
      set.Set(10, true);

      int index = 0;
      foreach (bool value in set)
      {
         Console.WriteLine("set[{0}] = {1}", index, value);
         index++;
      }
   }
}
```

Резултатът от изпълнение на примера е следният:



Как работи примерът?

Класът **BitSet32** представлява множество от 32 булеви стойности. Той съхранява стойностите си в **UInt32** поле като комбинация от битове – по 1 бит за всяка от тях. Методът **Set(index, value)** изчислява битова маска за зададения индекс, нулира съответния бит и ако е зададена стойност **true**, го установява след това в единица. Методът **Get(index)** изчислява битова маска за зададения индекс и връща стойността на съответния бит.

Класът **BitSet32** имплементира интерфейса **IEnumerable** като в метода му **GetEnumerator()** създава и връща инстанция на специален вътрешен клас **BitSet32Enumerator**, инициализирана по текущия **BitSet32** обект.

Класът **BitSet32Enumerator** е имплементация на интерфейса **IEnumerator**. Той съхранява текущия индекс от обхождането на **BitSet32** обекта във вътрешна променлива **mCurrentIndex**. Методът **MoveNext()** увеличава текущия индекс и ако не е достигнат краят, връща **true**. Методът **Reset()**

задава стойност **-1** за текущия индекс (това е елементът преди първия). Свойството **Current** връща елемента от текущата позиция.

Главната програма демонстрира правилната работа на класа **BitSet32** и имплементацията на интерфейсите **IEnumerable** и **IEnumerator**. Тя създава инстанция на **BitSet32**, променя някои от битовете и отпечатва всички стойности с цикъл **foreach**.

Упражнения

1. Избройте основните разлики между стойностните и референтните типове. Кои от следните типове са стойностни и кои референтни?
   - **int**, **char**, **string**, **float**, изброени типове, класове, структури, интерфейси, делегати, масиви, указатели, опаковани стойностни типове
2. Дефинирайте клас **Student**, който съдържа данните за един студент: трите му имена, ЕГН, местоживеене (постоянен и временен адрес), телефон (стационарен и мобилен), e-mail, курс, специалност, ВУЗ, факултет и т.н. Използвайте изброен тип (enumeration) за специалностите, ВУЗ-овете и факултетите. Реализирайте стандартните методи, наследени от **System.Object**: **Equals(object)**, **ToString()**, **GetHashCode()** и операторите **==** и **!=**.
3. Добавете имплементация на интерфейса **ICloneable** за класа **Student**. Методът **Clone()** трябва да копира в нов обект всяко от полетата на класа **Student**.
4. Дефинирайте структурата от данни двоично наредено дърво за претърсване (binary search tree) с операции "добавяне на елемент", "търсене на елемент" и "изтриване на елемент". Не е необходимо да поддържате дървото балансирано (това ще ви спести много усилия). Имплементирайте виртуалните методи **ToString()**, **Equals(object)**, **GetHashCode()** от **System.Object** и операторите за сравнение **==** и **!=**. Добавете и реализация на интерфейса **ICloneable** за дълбоко копиране на дървото.

   Упътване: За да улесните работата си, използвайте два типа – клас **BinarySearchTree** (за самото дърво) и клас **TreeNode** (за елементите на дървото).
5. Дефинирайте клас **ComplexNumber**, който съдържа комплексно число. Имплементирайте за него интерфейса **IComparable**.
6. Дефинирайте клас **BitSet256**, който представлява масив от 256 булеви стойности и се съхранява вътрешно като 4 на брой 64-битови полета (**UInt64**). Реализирайте методи **Get(int index)**, **Set(int index, bool value)** и индексатор за достъп. Имплементирайте и интерфейса **IEnumerable**, като за целта използвате вътрешен клас, който имплементира **IEnumerator**.

Използвана литература

1. Светлин Наков, Обща система от типове (Common Type System) – http://www.nakov.com/dotnet/lectures/Lecture-4-Common-Type-System-v1.0.ppt
2. Jeffrey Richter, Applied Microsoft .NET Framework Programming, Microsoft Press, 2002, ISBN 0735614229
3. Tom Archer, Andrew Whitechapel, Inside C#, 2-nd Edition, Microsoft Press, 2002, ISBN 0735616485
4. MSDN Training, Programming with the MSicrosoft® .NET Framework (MOC 2349B), Module 5: Common Type System
5. Svetlin Nakov, .NET Framework Overview – http://www.nakov.com/publications/Nakov-DotNET-Framework-Overview-english.ppt
6. MSDN Library – http://msdn.microsoft.com

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 6. Делегати и събития

Необходими знания

- Базови познания за архитектурата на .NET Framework
- Базови познания за общата система от типове в .NET (Common Type System)
- Базови познания по обектно-ориентирано програмиране с .NET Framework и C#

Съдържание

- Делегати (delegates). Дефиниране, инстанциране, извикване
- Single-cast и multicast делегати
- Събития (events)
- Разлика между събитие и инстанция на делегат
- Утвърдени конвенции при дефиниране и използване на събития в .NET Framework
- Кога да използваме интерфейси, събития и делегати?

В тази тема ...

В настоящата тема ще разгледаме референтния тип делегат. Ще се запознаем с начините на неговото използване, различните видове делегати, както и негови характерни приложения. Ще представим понятието събитие и ще обясним връзката му с делегатите. Ще сравним делегатите и интерфейсите и ще видим в кои случаи е добре да се използват едните и в кои – другите.

Какво представляват делегатите?

Делегатите са референтни типове, които описват сигнатурата на даден метод (броя, типа и последователността на параметрите му) и връщания от него тип. Могат да се разглеждат като "обвивки" на методи - те представляват структури от данни, които приемат като стойност методи, отговарящи на описаната от делегата сигнатура и връщан тип.

Инстанциране на делегат

Делегатът се инстанцира като клас и методът се подава като параметър на конструктора. Възможно е делегатът да сочи към повече от един метод, но на това ще се спрем подробно малко по-нататък.

Делегатите и указателите към функции

Съществува известна прилика между делегатите и указателите към функции в други езици, например Pascal, C, C++, тъй като последните представляват типизиран указател към функция. Делегатите също съдържат силно типизиран указател към функция, но те са и нещо повече – те са напълно обектно-ориентирани. На практика делегатите представляват класове. Инстанцията на един делегат може да съдържа в себе си както инстанция на обект, така и метод.

Callback методи и делегати

Едно от основните приложения на делегатите е реализацията на "обратни извиквания", т.нар. callbacks. Идеята е да се предаде референция към метод, който да бъде извикан по-късно. Така може да се осъществи например асинхронна обработка – от даден код извикваме метод, като му подаваме callback метод и продължаваме работа, а извиканият метод извиква callback метода когато е необходимо. Със средствата на делегатите е възможно даден клас да позволи на потребителите си да предоставят метод, извършващ специфична обработка, като по този начин обработката не се фиксира предварително.

Статични или екземплярни методи

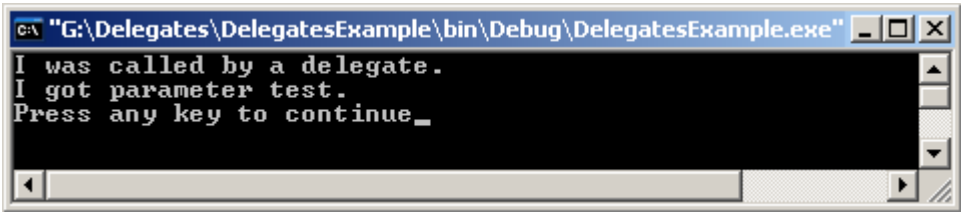
Делегатите могат да сочат както към методи на инстанция на класа, в който са декларирани, така и към статични методи. Това представлява удобство, защото можем да използваме делегат, без да сме създали инстанция на съдържащия го клас. Така се спестява създаването на допълнителна инстанция на клас. Друга възможност е да се отложи създаването на инстанция на делегат докато тя стане необходима. За целта можем да дефинираме свойство на класа, който ползва делегата, и в `get` метода на свойството да създадем делегата.

Пример за делегат

Следващият пример демонстрира деклариране на делегат, инстанциране на делегат и извикване на метод, сочен от него.

```
// Declaration of a delegate
public delegate void SimpleDelegate(string aParam);

class TestDelegate
{
   public static void TestFunction(string aParam)
   {
      Console.WriteLine("I was called by a delegate.");
      Console.WriteLine("I got parameter {0}.", aParam);
   }

   public static void Main()
   {
      // Instantiation of a delegate
      SimpleDelegate simpleDelegate =
         new SimpleDelegate(TestFunction);
      // Invocation of the method, pointed by a delegate
      simpleDelegate("test");
   }
}
```

След изпълнение на примера се получава следният резултат:



Описание на примера

В първия ред от кода се декларира делегат. За целта се използва ключовата дума **delegate**. След това в класа се дефинира функция, която има сигнатура и връщан тип като тези, декларирани от делегата. В главния метод на класа се инстанцира делегата, като дефинираният метод се подава като параметър и след това той се извиква чрез делегата.

Видове делегати

Делегатите в .NET Framework са специални класове, които наследяват **System.Delegate** или **System.MulticastDelegate**. От тези класове обаче явно могат да наследяват само CLR и компилаторът. Всъщност, те не са от тип делегат – тези класове се използват, за да се наследяват от тях типове делегат.

Всеки делегат има **"списък на извикване" (invocation list)**, който представлява наредено множество делегати, като всеки елемент от него съдържа конкретен метод, реферирaн от делегата. Делегатите могат да бъдат единични и множествени.

Единични (singlecast) делегати

Единичните делегати наследяват класа **`System.Delegate`**. Тези делегати извикват точно един метод. В списъка си на извикване имат единствен елемент, съдържащ референция към метод.

Множествени (multicast) делегати

Множествените делегати наследяват класа **`System.MulticastDelegate`**, който от своя страна е наследник на класа на **`System.Delegate`**. Те могат да викат един или повече метода. Техните списъци на извикване съдържат множество елементи, всеки реферирaщ метод. В тях може един и същ метод да се среща повече от веднъж. При извикване делегатът активира всички реферирани методи. Множествените делегати могат да участват в комбиниращи операции.

Езикът C# съдържа запазената дума **`delegate`**, чрез която се декларира делегат. При тази декларация компилаторът автоматично наследява типа **`MulticastDelegate`**., т.е. създава множествен делегат. Затова ще обърнем по-голямо внимание именно на този вид делегати.

На практика singlecast делегатите почти не се използват и под делегат обикновено се има предвид multicast делегат.

Извикване на multicast делегати

При извикване на multicast делегат се изпълняват всички методи от неговия списък на извикване. Методите се викат в реда, в който се намират в списъка, като дублиращите се методи (ако има такива) се викат толкова пъти, колкото се срещат в списъка.

Делегати и връщани стойности

Ако сигнатурата на методите, викани от делегата включва връщана стойност, връща се стойността, получена при изпълнението на последния елемент от списъка на делегата. Когато сигнатурата включва **`out`** или **`ref`** параметър, то всички извикани методи променят неговата стойност последователно в реда си на извикване и крайният резултат е резултата от последния извикан метод.

Делегати и изключения

Възможно е при извикване на multicast делегат някой от методите от списъка му на извикване да хвърли изключение. В този случай методът спира изпълнение и управлението се връща в кода, извикал делегата.

Останалите методи от списъка не се извикват. Дори извикващият метода да хване изключението, останалите методи от списъка не се изпълняват.

System.MulticastDelegate

Класът **System.MulticastDelegat** е наследник на **System.Delegate**. Той е базов клас за всички делегати в C#, но самият той не е тип делегат – при срещане на ключовата дума **delegate** компилаторът създава клас, наследник на **System.MulticastDelegat**. Всички делегати наследяват от него няколко важни метода, които сега ще разгледаме.

Комбиниране на делегати

Multicast делегатите могат да участват в комбиниращи операции. Това се реализира с метода **Combine()** на класа. Той слива списъците от методи на няколко делегата от еднакъв тип. Този метод е предефиниран и може да приема като параметри както два multicast делегата от еднакъв тип, така и масив от multicast делегати от еднакъв тип. В резултат връща нов multicast делегат, чийто списък от методи съдържа списъците на подадените като параметри делегати.

Неизменяемост на делегатите

Делегатът е неизменяем обект – веднъж създаден, повече не може да се променя.

Операцията "сливане" не променя съществуващите делегати, а създава нов делегат. Ако делегатът, получен в резултат на комбиниране не реферира нито един метод, **Combine()** връща стойност **null**, а не делегат с празен списък от методи. В C# е предефиниран операторът **+=** за комбиниране на делегати.

Метод Remove()

Освен, че списъците от методи на няколко делегата могат да бъдат обединявани в един, възможно е също от списъка на един делегат да се извади списъкът на друг. Това се извършва чрез метода **Remove()**. Той приема като параметри два делегата и в резултат връща нов делегат, чийто списък е получен като от списъка на първия аргумент е премахнато последното срещане на списъка на втория аргумент. Ако двата списъка са еднакви, или ако списъкът на втория аргумент не се среща в списъка на първия, резултатът е **null**. В езика C# е предефиниран операторът **-=** за изваждане на списъци на делегати.

Получаване на списъка от методи

С метода **GetInvocationList()** може да се получат методите, викани от делегата. По-точно, методът връща масив от делегати от типа на делегата, за който се вика методът. Всеки делегат от масива съдържа в списъка си от методи единствен елемент – някой от методите, викани от делегата. В

масива има по един делегат за всеки метод и делегатите са подредени така, както се извикват от multicast делегата. Извикването на делегатите последователно в реда, в който се срещат в масива, ще има същия ефект като от извикване на самия делегат.

Други методи

При деклариране на делегат компилаторът създава няколко служебни метода, които не могат да се извикват явно. Методът **`Invoke()`** извиква метода (методите), сочен (сочени) от делегата. Следователно при обръщение към делегата всъщност се извиква методът **`Invoke()`**, а той вика методите от списъка на делегата. Други методи са **`BeginInvoke()`** и **`EndInvoke()`**, чрез които се реализира асинхронно извикване. Освен това компилаторът декларира и конструктор на делегата.

Освен описаните методи, класът **`System.MulticastDelegate`** има и едно важно свойство – **`Method`**. То връща първия метод от списъка на делегата.

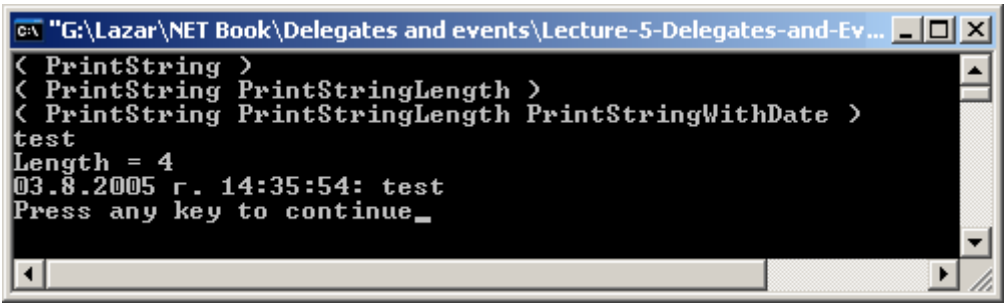
Multicast делегати – пример

В настоящия пример се демонстрира работата с multicast делегати и се илюстрира хода на изпълнение на програмния код:

```
using System;

public delegate void StringDelegate(string aValue);

public class TestDelegateClass
{
   void PrintString(string aValue)
   {
      Console.WriteLine(aValue);
   }

   void PrintStringLength(string aValue)
   {
      Console.WriteLine("Length = {0}", aValue.Length);
   }

   static void PrintStringWithDate(string aValue)
   {
      Console.WriteLine("{0}: {1}", DateTime.Now, aValue);
   }

   static void PrintInvocationList(Delegate aDelegate)
   {
      Console.Write("(");
      Delegate[] list = aDelegate.GetInvocationList();
      foreach (Delegate d in list)
      {
         Console.Write(" {0}", d.Method.Name);
```

```
      }
      Console.WriteLine(" )");
   }

   public static void Main()
   {
      TestDelegateClass tdc = new TestDelegateClass();
      StringDelegate printDelegate =
        new StringDelegate(tdc.PrintString);
      StringDelegate printLengthDelegate =
        new StringDelegate(tdc.PrintStringLength);
      StringDelegate printWithDateDelegate =
        new StringDelegate(PrintStringWithDate);

      PrintInvocationList(printDelegate);
      // Prints: ( PrintString )

      StringDelegate combinedDelegate = (StringDelegate)
        Delegate.Combine(printDelegate, printLengthDelegate);

      PrintInvocationList(combinedDelegate);
      // Prints: ( PrintString PrintStringLength )

      combinedDelegate = (StringDelegate)
        Delegate.Combine(combinedDelegate,
        printWithDateDelegate);

      PrintInvocationList(combinedDelegate);
      // Prints: ( PrintString PrintStringLength
      // PrintStringWithDate )

      // Invoke the delegate
      combinedDelegate("test");
   }
}
```

След изпълнение на примера се получава следният резултат:



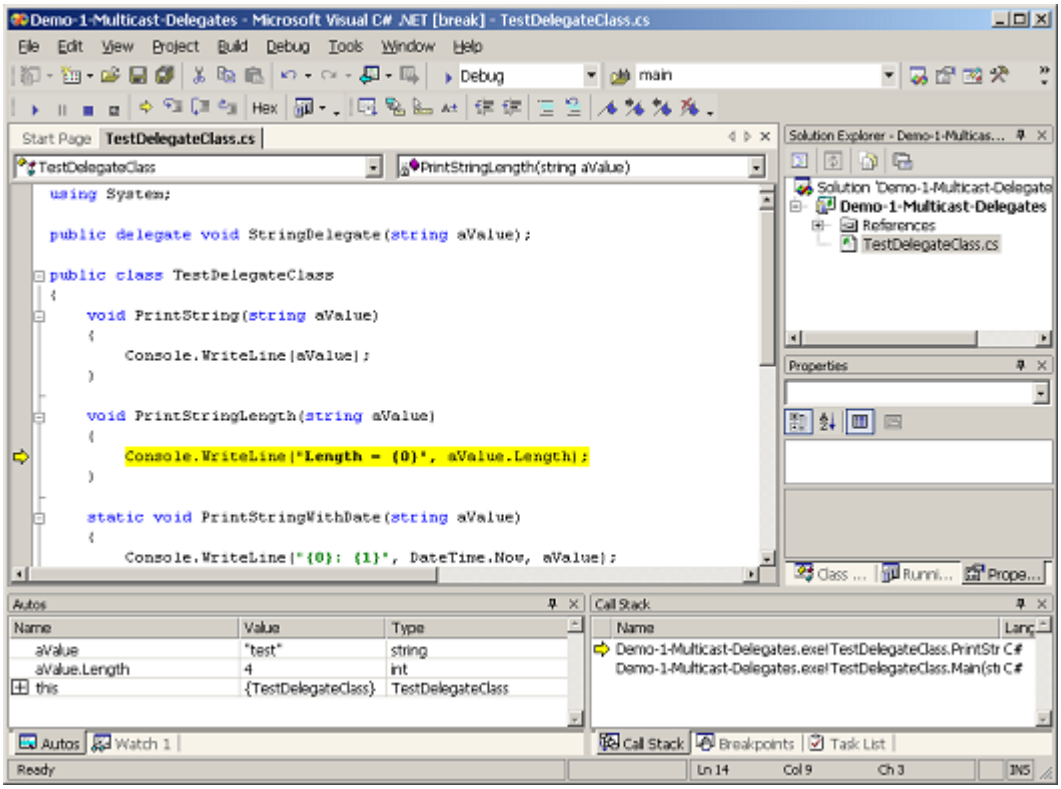
Описание на примера

Най-напред се декларира делегат, след което в класа се дефинират три метода, чиито сигнатури съответстват на декларираната от делегата. В главния метод на класа се създават три инстанции на делегата, всяка съдържаща референция към някой от трите метода и се извежда списъкът с методи на първата инстанция (той съдържа само един метод). След това се създава делегат, обединяващ списъците на първите два делегата, отпечатва се неговия списък с методи (този път списъкът съдържа два метода) и накрая към комбинирания делегат се добавя и третата инстанция на делегат. Отново се извежда списъкът от методи, който сега съдържа и трите метода, реферирани от делегатите, и трите метода се извикват посредством комбинирания делегат.

Проследяване на изпълнението на примера

За проследяване изпълнението на примера стъпка по стъпка ще използваме проекта **`Demo-1-Multicast-Delegates`** от демонстрациите, който съдържа кода от горния пример. Изпълняваме следните стъпки:

1. Отваряме проекта **`Demo-1-Multicast-Delegates.sln`**.
2. Слагаме точка на прекъсване на последния ред на **`Main()`** метода на основния клас, където става извикването на комбинирания делегат и стартираме приложението с **`[F5]`**.



3. Натискаме последователно **[F11]**, за да видим как трите метода, подадени при създаването на инстанциите на трите делегата, се извикват един след друг.

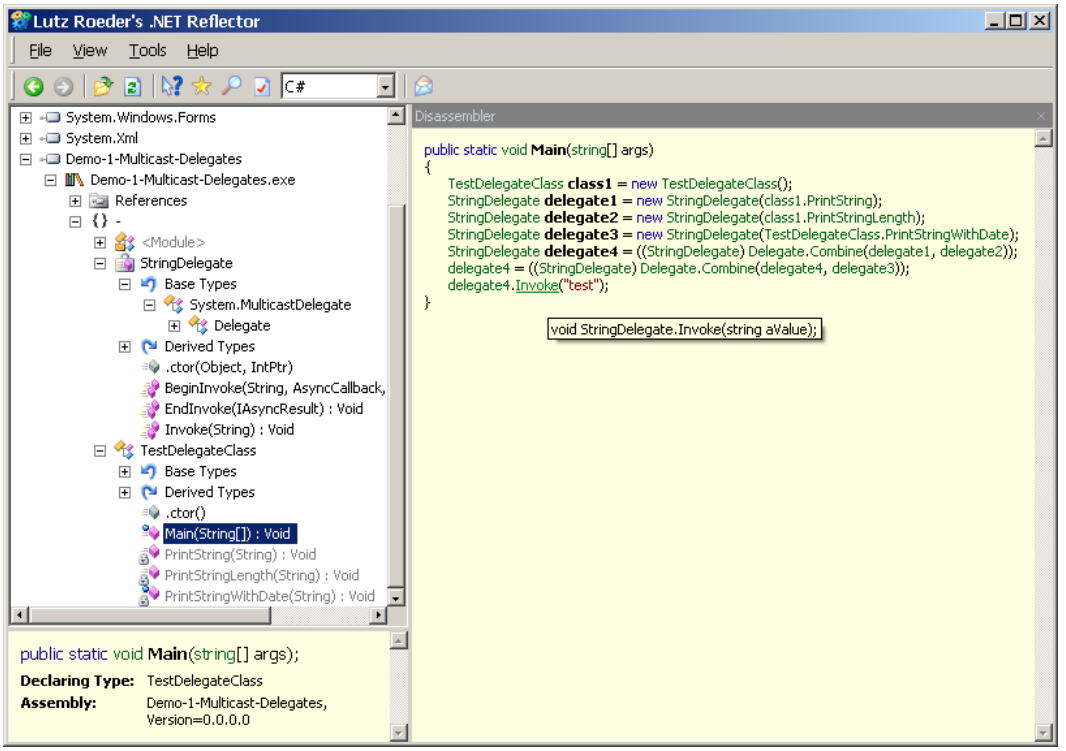
На картинката по-горе е показан изглед от VS.NET, в момент на постъпково проследяване на изпълнението на примера.

Инструментът .NET Reflector

Инструментът **.NET Reflector** е декомпилатор за .NET асемблита. Използва се за генериране на програмен код от изпълним код. Той има удобен потребителски интерфейс и за разлика от вградения в .NET Framework SDK инструмент ILDASM **.NET Reflector** може да декомпилира до код на C# и VB.NET, а не само до MSIL код. Инструментът е безплатен и може да бъде изтеглен от адрес http://www.aisto.com/roeder/dotnet/.

Използване на .NET Reflector

Настоящият пример илюстрира използването на инструмента .NET Reflector за разглеждане на кода, който компилаторът на C# генерира при деклариране на делегат.



За да декомпилираме асемблито от предишния пример и да разгледаме кода му, трябва да изпълним следните стъпки:

1. Стартираме **.NET Reflector**.

2. От менюто File | Open отваряме асемблито **`Demo-1-Multicast-Delegates.exe`**.
3. В дървото в лявата част на .NET Reflector намираме дефиницията на делегата **`StringDelegate`**. Вижда се, че той е обикновен клас, който наследява класа **`System.MulticastDelegate`** (това се вижда като се разгърне клона Base Types).
4. Намираме в дървото класа **`TestDelegateClass`**. Разгръщаме методите на класа и щракваме два пъти върху главния метод на класа. Разглеждаме генерирания при декомпилацията C# код. Можем да проверим, че при извикването на multicast делегата всъщност се извиква неговият **`Invoke()`** метод.

Събития (Events)

Събитията могат да се разглеждат като съобщения за настъпване на някакво действие. В компонентно-ориентираното програмиране компонентите изпращат събития (events) към своя притежател за да го уведомят за настъпването на интересна за него ситуация. Този модел е много характерен например за графичните потребителски интерфейси, където контролите уведомяват чрез събития други класове от програмата за действия от страна на потребителя. Например, когато потребителят натисне бутон, бутонът предизвиква събитие, с което известява, че е бил натиснат. Разбира се, събития могат да се предизвикват не само при реализиране на потребителски интерфейси. Нека вземем за пример програма, в която като част от функционалността влиза трансфер на файлове. Приключването на трансфера на файл може да се съобщава чрез събитие.

Шаблонът "Наблюдател"

Механизмът на събитията реализира шаблона "Наблюдател" (Observer) или, както още се нарича, Публикуващ/Абонати (Publisher/Subscriber), при който един клас публикува събитие, а произволен брой други класове могат да се абонират за това събитие. По този начин се реализира връзка на зависимост от тип един към много, при която когато един обект промени състоянието си, зависещите от него обекти биват информирани за промяната и автоматично се обновяват.

Изпращачи и получатели

Обектът, който предизвиква дадено събитие се нарича **изпращач на събитието (event sender)**. Обектът, който получава дадено събитие се нарича **получател на събитието (event receiver)**. За да могат да получават дадено събитие, получателите му трябва преди това да се абонират за него (subscribe for event).

За едно събитие могат да се абонират произволен брой получатели. Изпращачът на събитието не знае кои ще са получателите на събитието, което той предизвиква. Затова чрез механизма на събитията се постига

по-ниска степен на свързаност (coupling) между отделните компоненти на програмата.

Събитията в .NET Framework

В компонентния модел на .NET Framework абонирането, изпращането и получаването на събития се поддържа чрез делегати и събития. Реализацията на механизма на събитията е едно от главните приложения на делегатите. Класът, който публикува събитието, дефинира делегат, който абонатите на събитието трябва да имплементират. Когато събитието бъде предизвикано, методите на абонатите се извикват посредством делегата. Тези методи обикновено се наричат **обработчици** на събитието. Делегатът е multicast делегат, за да могат чрез него да се извикват много обработващи методи (на всички абонати).

Деклариране на събития

В C# събитията представляват специални инстанции на делегати. Те се декларират с ключовата дума **`event`**, която може да се предшества от модификатори, като например модификатори за достъп. Обикновено събитията са с модификатор **`public`**. След ключовата дума **`event`** се записва името на делегата, с който се свързва съответното събитие. За тази цел делегатът трябва да бъде дефиниран предварително. Той може да бъде дефиниран от потребителя, но може да се използва и вграден делегат. Тези делегати трябва да бъдат от тип **`void`**.

Операции върху събития

Единствените операции, които са позволени върху събития са операциите за абониране и премахване на абонамент. За целта при деклариране на събитие компилаторът автоматично дефинира операторите += за абониране за събитие и -= за премахване на абонамент. Тези оператори могат да бъдат извиквани от класове, външни за класа, в който е дефинирано събитието, така че външен код може да добавя и премахва обработчици на събитието, но не може по никакъв друг начин да манипулира списъка с обработчици. Възможно е подразбиращото се поведение на операторите += и -= да бъде предефинирано.

Разлика между събитие и делегат

Събитията и делегатите са много тясно свързани. Въпреки това член-променлива от тип делегат не е еквивалентна на събитие, декларирано с ключовата дума event, т.е. **`public MyDelegate m`** не е същото като **`public event MyDelegate m`**. Първото е декларация на променлива **`m`**, която е от тип **`MyDelegate`**, докато второто декларира събитие, което ще се обработва от делегат от тип **`MyDelegate`**.

Между делегатите и събитията има и други разлики, освен в деклариране-то. Например, делегатите не могат да бъдат членове на интерфейси,

докато събитията могат. В такъв случай класът, който имплементира интерфейса, трябва да предостави подходящо събитие.

Извикване на събитие

Извикването на събитие може да стане само в класа, в който то е дефинирано. Това означава, че само класът, в който се дефинира събитие, може да предизвика това събитие. Това е наложително, за да се спази шаблонът на Публикуващ/Абонати – абонираните класове се информират при промяна на състоянието на публикуващия и именно публикуващият е отговорен за разпращане на съобщенията за промяната, настъпила у него.

Друга важна подробност за събитията е, че достъпът до тях е синхронизиран. Това има значение при създаването на многонишкови приложения, с които ще се запознаем подробно в темата "Многонишково програмиране и синхронизация".

Конвенция за събитията

В .NET Framework се използва утвърдена конвенция за събитията. Тя определя именуването на събитията и свързаните с тях методи и делегати, връщаните типове и приеманите аргументи от делегатите.

Делегати за събития

Делегатите, използвани за събития по конвенция имат имена, които се състоят от глагол и **`EventHandler`** (**`SomeVerb`*`EventHandler`***). Така те се различават лесно от декларациите на други делегати в приложението.

Делегатите, използвани за събития, не трябва да връщат стойност. Връщаният тип от делегата трябва да бъде `void`.

Аргументи на делегата

Конвенцията налага делегатите, които ще се използват от събития, да приемат два аргумента. Единият представлява обектът-изпращач на събитието, т. е. това е източникът на събитието, или публикуващият от шаблона Публикуващ/Абонати. Той трябва да е от тип **`System.Object`**. Другият аргумент представя информация за изпращаното събитие. Той е от тип, наследник на **`System.EventArgs`**.

Ето пример за деклариране на делегат, който ще бъде използван от събитие:

```
public delegate void ItemChangedEventHandler(
  object aSender, ItemChangedEventArgs aEventArgs);
```

Конвенция за деклариране на събитията

Събитията обикновено се обявяват като **public**, въпреки че са възможни и останалите модификатори за достъп. Имената им започват с главна буква, а последната дума от името е глагол. Следва пример за декларация на събитие:

```
public event ItemChangedEventHandler ItemChanged;
```

Предизвикване на събитие

За предизвикване на събитие се създава **protected void** метод. Прието е името му да започва с **On**, следвано от името на събитието (например **OnEventName**). Този метод предизвиква събитието като извиква делегата.

Методът трябва да е **protected**, защото това позволява при наследяване на класа, в който е декларирано събитието, наследниците да могат да предизвикват събитието. Ако методът не е **protected**, наследниците няма да могат да предизвикат събитието, защото не могат да се обърнат директно към него, тъй като то е достъпно единствено в класа, в който е декларирано. За още по-голяма гъвкавост е възможно освен **protected**, методът да бъде обявен **virtual**, което би позволило на наследниците да го предефинират. Така те биха могли да прихващат извикването на събитията от базовия клас и евентуално да извършват собствена обработка. Следващият пример показва как се декларира метод за предизвикване на събитие:

```
protected void OnItemChanged() { … }
```

Конвенция за обработчиците

Обикновено името на метода-получател (обработчикът) на събитието има вида Обект_Събитие, както се илюстрира в следния пример:

```
private void OrderList_ItemChanged () { … }
```

Събития – пример

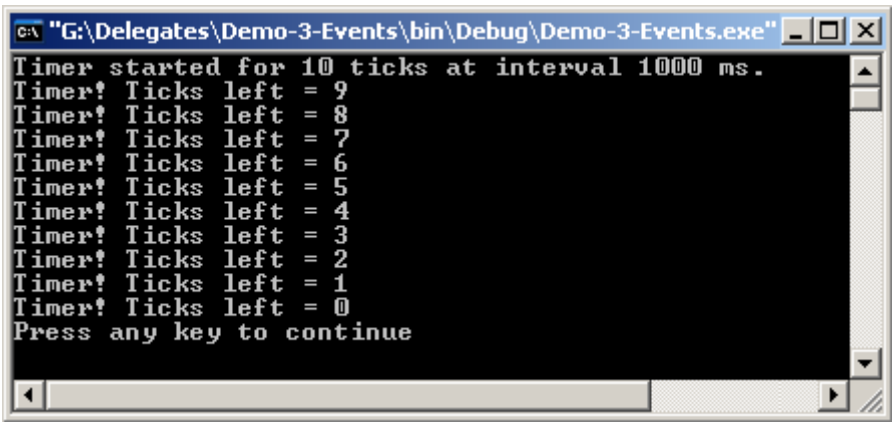
В настоящия пример се разглежда дефинирането и използването на събития като се спазва утвърдената конвенция в .NET Framework. Демонстрира се изпращане и получаване на събития.

```
// A delegate type for hooking up change notifications
public delegate void TimeChangedEventHandler(
   object aSender, TimeChangedEventArgs aEventArgs);

// A class that inherits System.EventArgs and adds
// information for the current time
```

```
public class TimeChangedEventArgs : EventArgs
{
   private int mTicksLeft;

   public TimeChangedEventArgs(int aTicksLeft)
   {
      mTicksLeft = aTicksLeft;
   }

   public int TicksLeft
   {
      get
      {
         return mTicksLeft;
      }
   }
}

public class Timer
{
   private int mTickCount;
   private int mInterval;

   // The event that will be raised when the time changes
   public event TimeChangedEventHandler TimeChanged;

   public Timer(int aTickCount, int aInterval)
   {
      mTickCount = aTickCount;
      mInterval = aInterval;
   }

   public int TickCount
   {
      get
      {
         return mTickCount;
      }
   }

   public int Interval
   {
      get
      {
         return mInterval;
      }
   }

   // The method that invokes the event
   protected void OnTimeChanged(int aTick)
```

```
   {
      if (TimeChanged != null)
      {
         TimeChangedEventArgs args =
           new TimeChangedEventArgs(aTick);
         TimeChanged(this, args);
      }
   }

   public void Run()
   {
      int tick = mTickCount;
      while (tick > 0)
      {
         System.Threading.Thread.Sleep(mInterval);
         tick--;
         OnTimeChanged(tick);
      }
   }
}

public class TimerDemo
{
   // The event handler method
   private static void Timer_TimeChanged(object aSender,
      TimeChangedEventArgs aEventArgs)
   {
      Console.WriteLine("Timer! Ticks left = {0}",
         aEventArgs.TicksLeft);
   }

   public static void Main()
   {
      Timer timer = new Timer(10, 1000);
      timer.TimeChanged +=
         new TimeChangedEventHandler(Timer_TimeChanged);
      Console.WriteLine(
         "Timer started for 10 ticks at interval 1000 ms.");
      timer.Run();
   }
}
```

При изпълнение на примера се получава следният резултат:



Описание на примера

Класът **Timer** от примера служи за предизвикване на дадено събитие през определен интервал от време.

Най-напред се декларира делегатът **TimeChangedEventHandler**, който ще бъде типа на предизвиканото събитие. Според обяснените конвенции той приема два аргумента – един от тип **Object**, и един от тип, наследник на **EventArgs**. Този наследник е класът **TimeChangedEventArgs**. Той добавя член, който съдържа информация за оставащия брой извиквания.

В класа **Timer** се декларира събитието **TimeChanged** от тип **TimeChangedEventHandler**. Методът **OnTimeChanged** на класа **Timer** проверява дали има абонати за събитието (за целта проверява дали събитието няма стойност **null**) и в случай, че има абонати предизвиква събитието. В метода **Run()** на класа **Timer** периодично се предизвиква събитието **TimeChanged** чрез обръщение към метода **OnTimeChanged**.

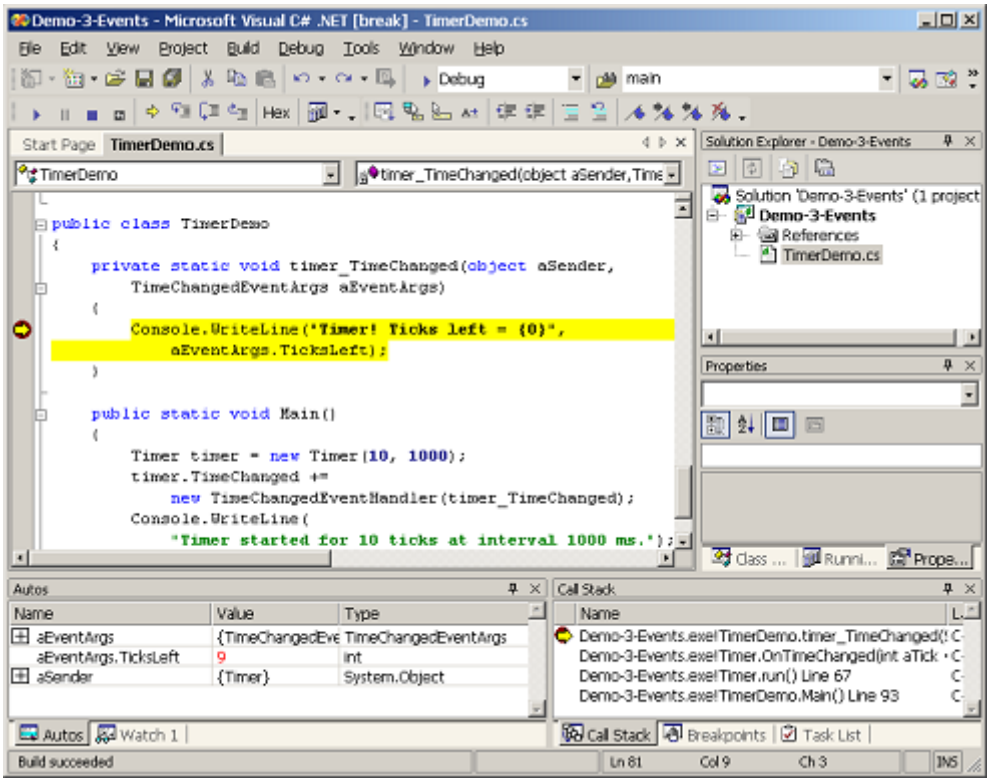
В класа **TimerDemo** се декларира обработчик на събитието **TimeChanged** и в главната функция **TimerDemo** се абонира за събитието.

Проследяване на изпълнението на примера

За проследяване на примера стъпка по стъпка можем да използваме проекта **Demo-3-Events** от демонстрациите. За целта изпълняваме следните стъпки:

1. Отваряме проекта **Demo-3-Events.sln**. Той съдържа горния пример.
2. Слагаме точки на прекъсване в методите **timer_TimeChanged()** и **OnTimeChanged()**.
3. Стартираме приложението с **[F5]**.
4. Натискаме след това последователно **[F11]**, за да видим как през една секунда се изпраща и съответно получава събитието **TimeChanged**.

На картинката е показан изглед от VS.NET в момент на постъпково проследяване на изпълнението на примера:



Делегатът System.EventHandler

Не винаги има нужда събитието да генерира някакви данни, които да изпрати на абонатите. В такъв случай може да се използва вградения делегат **`System.EventHandler`**. Той дефинира референция към callback метод, който обработва именно такива събития. Съответно методите-обработчици на такива събития трябва да съответстват на декларираните от делегата **`System.EventHandler`** връщан тип и сигнатура. Следва декларацията на делегата:

```
public delegate void EventHandler(Object sender, EventArgs e);
```

Този делегат се използва на много места вътрешно от .NET Framework. Например, типът на събитието, което възниква при натискане на бутон, е точно **`EventHandler`**, тъй като при това събитие не се генерира информация за предаване.

Класът EventArgs

Както се вижда от горната дефиниция, вграденият делегат **`EventHandler`** има като втори аргумент обект от тип **`EventArgs`**. За този тип вече стана

дума – всеки делегат, който се използва за тип на събитие трябва да приема аргумент от тип, който е наследник на **EventArgs**.

Класът **EventArgs** наследява всичките си членове от **System.Object**, като добавя конструктор и публично статично поле **Empty**, което представя събитие без данни. По този начин се улеснява използването на събития, които не носят данни.

Ако събитието трябва да съдържа информация се използва клас, който наследява **System.EventArgs** и добавя необходимите членове за нейното представяне. Ако все пак няма нужда от подобна информация, може директно да се използва инстанция на **System.EventArgs** при предизвикване на събитието. Случаят с **EventHandler** е точно такъв, тъй като той се използва за обработка на събития, които не носят информация. Затова той приема като аргумент директно **EventArgs**.

Пример за използване на System.EventHandler

В настоящия пример ще се илюстрира употребата на вградения делегат **System.EventHandler**:

```
public class Button
{
   public event EventHandler Click;
   public event EventHandler GotFocus;
   public event EventHandler TextChanged;
   ...
}

public class ButtonTest
{
   private static void Button_Click(object aSender,
      EventArgs aEventArgs)
   {
      Console.WriteLine("Button_Click() event called.");
   }

   public static void Main()
   {
      Button button = new Button();
      button.Click += new EventHandler(Button_Click);
      button.DoClick();
   }
}
```

Описание на примера

В горния пример се дефинира клас **Button** с набор събития. Класът **ButtonTest** дефинира функция **Button_Click(…)**, съответстваща на делегата за обработване на събитието **Click** на класа **Button** и в главния си

метод се абонира за това събитие и го предизвиква с обръщение към метода **DoClick()**, който за краткост е изпуснат в примера.

Събития и интерфейси

Както беше вече споменато, за разлика от делегатите, събитията могат да бъдат членове на интерфейси. Следващият код илюстрира това:

```
public interface IClickable
{
   event ClickEventHandler Click;
}
```

Имплементиране на събития в интерфейс

При имплементация на интерфейса, имплементиращият клас трябва да предизвиква събитието, което е декларирано в интерфейса. Допустимо е освен това при конкретната имплементация класът да реализира специфични **add** и **remove** методи, с което да промени тяхното поведение по подразбиране и да добави нетривиална логика.

Когато в интерфейс се декларират свойства, имплементиращият клас е длъжен да реализира техните методи (**set**, **get** или и двата, в зависимост от декларацията). За разлика от свойствата, при събитията не е задължително да се имплементират специфични **add** и **remove** методи – ако специфична реализация липсва, те получават поведение по подразбиране, съответно да добавят и премахват обработчици на събитието.

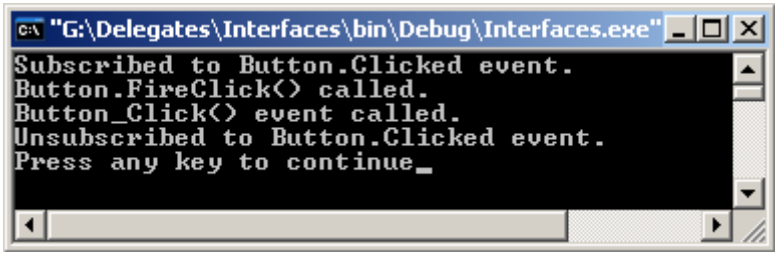
Събития и интерфейси – пример

В настоящия пример ще разгледаме съвместната употреба на събития и интерфейси:

```
public delegate void ClickEventHandler(object aSender,
   EventArgs aEventArgs);

public interface IClickable
{
   event ClickEventHandler Click;
}

public class Button : IClickable
{
   private ClickEventHandler mClick;

   // Implement the event from the interface IClickable
   public event ClickEventHandler Click
   {
      add
```

```
      {
         mClick += value;
         Console.WriteLine("Subscribed to Button.Clicked event.");
      }
      remove
      {
         mClick -= value;
         Console.WriteLine(
            "Unsubscribed from Button.Clicked event.");
      }
   }

   protected void OnClick()
   {
      if (mClick != null)
      {
         mClick(this, EventArgs.Empty);
      }
   }

   public void FireClick()
   {
      Console.WriteLine("Button.FireClick() called.");
      OnClick();
   }
}

public class ButtonTest
{
   private static void Button_Click(object aSender,
      EventArgs aEventArgs)
   {
      Console.WriteLine("Button_Click() event called.");
   }

   public static void Main()
   {
      Button button = new Button();
      button.Click +=
         new ClickEventHandler(Button_Click);
      button.FireClick();
      button.Click -=
         new ClickEventHandler(Button_Click);
   }
}
```

След изпълнение на примера се получава следният резултат:



Описание на примера

В началото се декларира делегат, който ще бъде тип на събитието, дефинирано в интерфейса. След това се дефинира интерфейс **`IClickable`** с единствен член, който е събитието **`Click`**. Класът **`Button`** имплементира интерфейса **`IClickable`**, като добавя частна член-променлива от типа на делегата и реализира специфични **`add`** и **`remove`** методи за събитието, чрез които се извършва манипулация на списъка с методи на променливата-делегат. Освен това в класа се декларират методи за предизвикване на събитието и извикване на обработчиците му. Класът **`ButtonTest`** дефинира метод-обработчик на събитието **`Click`** и в главния си метод създава инстанция на класа **`Button`**, абонира се за събитието **`Click`**, предизвиква го и след това премахва абонамента.

Интерфейси, събития, делегати

В .NET Framework поведението "обратно извикване" може да се реализира чрез три механизма: делегати, събития и интерфейси. Досега разгледахме използването на делегатите и събитията.

"Обратно извикване" чрез интерфейси

Чрез интерфейси "обратно извикване" може да се реализира като методът, който трябва да се използва за "обратно извикване" се декларира като член на интерфейс. След това в различните класове, които имплементират интерфейса, методът може да бъде имплементиран по различни начини и така той не се обвързва с конкретна реализация. В класа, който ще извършва обръщение към callback метода се декларира променлива от типа на интерфейса, който съдържа декларацията на съответния метод. На тази променлива могат да се присвояват референции към обекти от различни класове, имплементиращи съответния интерфейс. По този начин могат да бъдат викани методи с различно поведение, в зависимост от необходимостта.

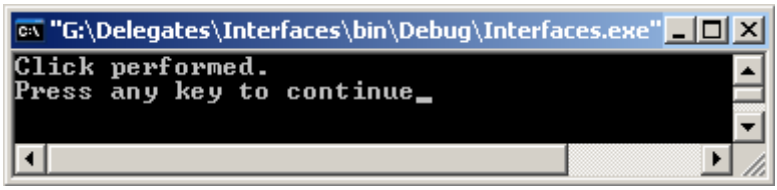
"Обратно извикване" чрез интерфейси – пример

Следващият пример показва как можем да използваме интерфейс, за да реализираме "обратно извикване":

```
public interface IClickListener
{
```

```
   void ClickPerformed();
}

public class Button
{
   private IClickListener mClickListener;

   public Button(IClickListener aClickListener)
   {
      mClickListener = aClickListener;
   }

   public void DoClick()
   {
      if (mClickListener != null)
      {
         mClickListener.ClickPerformed();
      }
   }
}

public class ButtonTest : IClickListener
{
   public static void Main()
   {
      ButtonTest buttonTest = new ButtonTest();
      Button button = new Button(buttonTest);
      button.DoClick();
   }

   void IClickListener.ClickPerformed()
   {
      Console.WriteLine("Click performed.");
   }
}
```

След изпълнения на примера се получава следният резултат:



Описание на примера

Интерфейсът **IClickListener** съдържа метода **ClickPerformed()**, чрез който ще се реализира "обратно извикване". В класа **Button** са дефинирани член **mClickListener** от типа на интерфейса и метод **DoClick()**, който извиква интерфейсния метод **ClickPerformed()**. Класът **ButtonTest**

имплементира интерфейса **`IClickListener`** и в главната си функция създава обект от тип **`Button`** и извиква метода му **`DoClick()`**. При това се извиква имплементацията на **`ClickPerformed()`**, която е предоставена от **`ButtonTest`**.

Кога да използваме интерфейси?

Въпреки че "обратно извикване" може да се реализира чрез интерфейси, това не е типично приложение на интерфейс и е добре да се използва по-рядко. Докато делегатите дават възможност за извикване на множество методи, то всички те трябва да имат еднакъв връщан тип и сигнатура. При интерфейсите няма такова ограничение, и затова те трябва да се ползват именно когато е нужно даден обект да предоставя съвкупност от много различни callback методи.

Кога да използваме събития?

Събитията се използват, когато разработваме компоненти, които трябва да известяват своя притежател за нещо, обикновено за промяна в текущото състояние или за извършване на някакво действие. Освен това чрез използване на събитията се поддържа съвместимост с компонентния модел на .NET.

Кога да използваме делегати?

Основните приложения на делегатите са за асинхронна обработка посредством callback методи и за даване на възможност на потребителите на клас да предоставят метод, извършващ специфична обработка, която не е предварително фиксирана. В този случай извикването на callback метода не е свързано с настъпването на някаква промяна или събитие, както е при събитията. Делегатите се използват, когато имаме единичен callback метод, който не е свързан с компонентния модел на .NET.

Упражнения

1. Обяснете какво представляват делегатите в .NET Framework.
2. Обяснете какво представляват събитията (events) в .NET Framework.
3. Какво се препоръчва от утвърдената конвенция за събитията в .NET Framework? Опишете програмния код за дефиниране и използване на събития.
4. Чрез средствата на делегатите реализирайте универсален статичен метод за изчисляване с някаква точност на безкрайни сходящи редове по зададена функция за общия им член. Чрез подходящи функции за общия член изчислете с точност два десетични знака безкрайните редове:
   - 1 + 1/2 + 1/4 + 1/8 + 1/16 + …
   - 1 + 1/4 + 1/9 + 1/16 + 1/25 + …

- 1 + 1/2! + 1/3! + 1/4! + 1/5! + …

5. Напишете клас **Person**, който описва един човек и съдържа свойствата: име, презиме, фамилия, пол, рождена дата, ЕГН, адрес, e-mail и телефон. Добавете към класа **Person** за всяко от неговите свойства по едно събитие от системния делегат **System.EventHandler**, което се активира при промяна на съответното свойство.
6. Създайте клас **PropertyChangedEventArgs**, наследник на класа **System.EventArgs** и дефинирайте в него три свойства – име на променено свойство (**string**), стара стойност (**object**) и нова стойност (**object**) заедно с подходящ конструктор. Създайте делегат **PropertyChangedEventHandler** за обработка на събития, който да приема два параметъра – обект-изпращач и инстанция на **PropertyChangedEventArgs**.
7. Напишете нов вариант на класа **Person**, който има само едно събитие с име **PropertyChanged** от тип **PropertyChangedEventHandler**, което се активира при промяна на някое от свойствата на класа (и съответно се извиква с подходящи параметри).
8. Изнесете дефиницията на събитието **PropertyChanged** в отделен интерфейс и променете класа така, че да имплементира интерфейса.

Използвана литература

1. Светлин Наков, Делегати и събития – http://www.nakov.com/dotnet/lectures/Lecture-5-Delegates-and-Events-v1.0.ppt
2. Jeffrey Richter, Applied Microsoft .NET Framework Programming, Microsoft Press, 2002, ISBN 0735614229
3. Jesse Liberty, Programming C#, O'Reilly, 2001, ISBN 0-596-00117-7
4. Andrew Whitechapel, Tom Archer, Inside C#, Microsoft Press, 2002, ISBN 0-7356-1648-5
5. MSDN Training, Programming with the Microsoft® .NET Framework (MOC 2349B), Module 8: Delegates and Events
6. Julien Couvreur, Curiosity is bliss – http://blog.monstuff.com/archives/000040.html
7. MSDN Library – http://msdn.microsoft.com

Глава 7. Атрибути

Необходими знания

- Базови познания за архитектурата на .NET Framework
- Базови познания за общата система от типове в .NET (Common Type System)
- Базови познания за езика C#

Съдържание

- Какво представляват атрибутите?
- Прилагане на атрибути. Атрибути с параметри. Задаване на цел при прилагане на атрибут
- Къде се използват атрибутите?
- Дефиниране на собствени атрибути
- Извличане на атрибути от метаданните
- Мета-атрибутът **`AttributeUsage`**

В тази тема ...

В настоящата тема ще разгледаме какво представляват атрибутите в .NET Framework, как се прилагат и къде се използват. Ще обясним как могат да се дефинират собствени атрибути и да се извличат атрибути от метаданните на асемблитата.

Какво представляват атрибутите в .NET?

В повечето езици за програмиране съществуват ключови думи. Такива например са спецификаторите за достъп, които определят областта на видимост на член-променливите на класовете (**`public`**, **`private`**, **`protected`**, …). Най-често компилаторите разпознават само ограничен набор ключови думи и програмистите нямат възможност да дефинират свои собствени.

За компенсиране на тази слабост в .NET Framework се дава възможност програмно да се добавят т. нар. **атрибути**. Те представляват описателни декларации към типове, полета, методи, свойства и други елементи на кода, подобни на ключовите думи от езиците за програмиране.

Атрибутите позволяват да се добавят собствени описателни елементи (анотации) към кода, написан на C# или на някой от другите езици от .NET платформата, без да се налага промяна в компилатора. По време на компилация те се записват в метаданните на асемблито и при изпълнение на кода могат да бъдат извличани и да влияят на поведението му.

Атрибутите са описателни тагове, които могат да се прилагат към различни елементи от кода, наричани **цели**. Целите могат да бъдат най-разнообразни: асемблита, типове, свойства, полета, методи, параметри и други елементи от кода.

Декларативна информация се асоциира с програмния код (към типовете, методите, свойствата и т.н.) чрез атрибути. Други приложения могат да извличат тази информация, за да определят как да бъдат използвани елементите, свързани с атрибутите.

Атрибутите реално представляват класове, които се инстанцират по време на компилация на сорс кода и се записват в метаданните на асемблито, от където могат да бъдат извличани по време на работа на приложението.

Атрибутите се делят на две групи – вградени в .NET Framework (които са част от CLR) и дефинирани от програмистите за целите на отделните приложения. Последните се наричат собствени (потребителски) атрибути и най-често се използват в комбинация с reflection (отражение на типовете).

Прилагане на атрибути

По-долу ще разгледаме начините, по които можем да приложим атрибут към дадена цел.

За да се приложи атрибут, името му се огражда в квадратни скоби и се поставя непосредствено преди декларацията, за която се отнася. Ето един пример:

```
// Apply attribute System.FlagsAttribute to FileAccess enum
```

```
[Flags]
public enum FileAccess
{
   Read = 1,
   Write = 2,
   ReadWrite = Read | Write
}
```

В посочения пример системният атрибут **Flags** (реално това е типът **System.FlagsAttribute**) е приложен към дефиницията на изброения тип **FileAccess** и указва, че този изброен тип може да се третира като битово поле, т.е. като множество от битови флагове.

За да бъде приложен атрибут към дадена дефиниция в кода, трябва да се изпълнят следните стъпки:

1. Да се дефинира нов атрибут или да се използва съществуващ, като неговото пространство от имена (namespace) се импортира в началото на текущия файл от сорс кода.
2. Да се изпише името на атрибута в квадратни скоби точно преди целта, към която се прилага. По желание могат да му бъдат предадени някакви параметри (инициализиращи данни).

Атрибутите за дадена цел могат да се прилагат и в комбинация. Това става по два начина: като се приложат един след друг или като се изброят със запетаи:

```
[MyFirstAttribute]
[MySecondAttribute]
public void SomeMethod() { … }

[MyFirstAttribute, MySecondAttribute]
public void SomeMethod() { … }
```

Двете декларации в горния пример са напълно еквивалентни. Те дефинират публичен метод **SomeMethod()** и прилагат към него атрибутите **MyFirstAttribute** и **MySecondAttribute**.

При прилагането на атрибути суфиксът **Attribute** може да бъде пропуснат и се подразбира от компилатора. Така следните декларации са еквивалентни на горните две:

```
[MyFirst]
[MySecond]
public void SomeMethod() { … }

[MyFirst, MySecond]
public void SomeMethod() { … }
```

Тъй като атрибутите са класове, при тяхното прилагане може да бъде извикван конструкторът на съответния клас. Ако атрибутът предлага конструктор без параметри, той може да бъде извикан като се добави **()** към декларацията. Следователно следващите две декларации са валидни и еквивалентни на предходните две:

```
[MyFirst()]
[MySecondAttribute()]
public void SomeMethod() { … }

[MyFirstAttribute(), MySecond()]
public void SomeMethod() { … }
```

От примерите виждаме, че има много синтактично валидни начини за прилагане на един и същ атрибут към дадена цел. За компилатора няма значение кой от тези варианти е употребен, но препоръката е да се използва този без суфикс **Attribute**, без изброяване със запетаи и без скоби **()**. За нашия пример препоръчителен е следният запис:

```
[MyFirst]
[MySecond]
public void SomeMethod() { … }
```

Атрибутите са обекти

Атрибутите в .NET Framework реално представляват .NET обекти (инстанции на клас, наследник на системния клас **System.Attribute**). Като такива те могат да имат един или няколко конструктора (вкл. конструктор по подразбиране), публични и частни полета, свойства и др. членове. Най-често атрибутите дефинират конструктори, публични полета и свойства, които използват за съхраняване на данните, подавани им като параметри по време на инициализация.

Всички атрибути в .NET Framework задължително наследяват класа **System.Attribute** (или негов наследник). Както ще видим по-долу, при дефиниране на собствени (потребителски атрибути) ние също трябва да наследяваме този клас.

Параметри на атрибутите

Някои атрибути могат да приемат параметри. Параметрите биват два вида: позиционни и именувани. Позиционните параметри се подават с определена последователност и се инициализират от конструктора на атрибута, докато именуваните се подават в произволен ред и задават стойност на свойство или публично поле. Ето един пример:

```
[DllImport("user32.dll", EntryPoint="MessageBox")]
public static extern int ShowMessageBox(int hWnd, string text,
```

```
   string caption, int type);
...
ShowMessageBox(0, "Some text", "Some caption", 0);
```

В примера е използвана комбинация между позиционни и непозиционни параметри. За да бъде приложен към метаданните в асемблито, атрибутът **[DllImport]** (**System.Runtime.InteropServices.DllImportAttribute**) по време на компилация се инстанцира и инициализира от компилатора по следния начин:

1. Създава се обект от класа **System.Runtime.InteropServices.DllImportAttribute**.
2. В конструктора му се подава като позиционен параметър стойност "**user32.dll**".
3. В публичното му поле **EntryPoint** се записва стойност "**MessageBox**".

Преди да бъдат записани в метаданните на съответното асембли, атрибутите се инициализират посредством подадените им параметри, с които се задават стойности за техните полета и свойства. При съхраняване в асемблито атрибутите запазват състоянието си.

На по-късен етап, когато атрибутите бъдат извлечени от метаданните на асемблито, стойностите на техните полета и свойства се извличат заедно с тях и могат да бъдат използвани от програмиста.

Задаване на цел към атрибут

Атрибутите в .NET Framework могат да се прилагат към различни цели, например асембли, клас, интерфейс, член-променлива на тип и др. Възможните цели на атрибутите се дефинират от изброения тип **AttributeTargets** както следва:

Име на целта	Употреба (прилага се към)
Assembly	самото асембли
Module	текущия модул
Class	клас
Struct	структура
Interface	интерфейс
Enum	изброен тип
Delegate	делегат
Constructor	конструктор
Method	метод
Parameter	параметър на метод

ReturnValue	връщаната стойност от метод
Property	свойство
Field	поле (член-променлива)
Event	събитие
All	всички възможни цели

При прилагане на атрибут целта обикновено се подразбира. Например, ако поставим атрибут преди декларацията на даден метод, той ще се отнася за съответния метод.

Понякога не може да се използва целта по подразбиране, например ако искаме да приложим атрибут към асемблито. В такива случаи целта може да се зададе преди името на атрибута, отделена от него с двоеточие:

```
// The following attributes are applied to the target "assembly"
[assembly: AssemblyTitle("Attributes Demo")]
[assembly: AssemblyCompany("DemoSoft")]
[assembly: AssemblyProduct("Entreprise Demo Suite")]
[assembly: AssemblyCopyright("(c) 1963-1964 DemoSoft")]
[assembly: AssemblyVersion("2.0.1.37")]

[Serializable] // The compiler assumes [type: Serializable]
class TestClass
{
   [NonSerialized] // The compiler assumes [field: NonSerialized]
   private int mStatus;
   ...
}
```

Както се вижда от коментарите в кода, за някои от атрибутите целта се задава експлицитно, а за други тя се подразбира.

Къде се използват атрибутите?

Атрибутите се използват вътрешно от .NET Framework за различни цели – при сериализация на данни, за описание на различни характеристики, свързани със сигурността на кода, за задаване на ограничения за оптимизациите от JIT компилатора, така че кодът да може да се дебъгва, за взаимодействие с дизайнера на средата за разработка при създаване на .NET компоненти, при взаимодействие с неуправляван код, при работа с уеб услуги, в ASP.NET потребителски контроли и на много други места.

Ще разгледаме някои от най-важните приложения на атрибутите вътрешно в .NET Framework.

Декларативно управление на сигурността

.NET Framework дава възможност на разработчиците да поставят ограничения върху сигурността в два стила. Императивният стил е свързан със създаването на **Permission** обекти по време на изпълнение и извършване на обръщения към техните методи. Декларативният се осъществява чрез атрибути. Следва пример за декларативно управление на сигурността чрез атрибути:

```
[PrincipalPermissionAttribute(SecurityAction.Demand,
   Name = "SomeUser", Role = "Administrator")]
public void DeleteCustomer(string aCustomerId)
{
   // Delete the customer
}
```

В горния пример чрез атрибута **PrincipalPermissionAttribute** се указва на CLR, че за изпълнението на метода **DeleteCustomer(…)** е необходимо текущата нишка да се изпълнява от потребител **SomeUser**, който е в роля **Administrator**.

В темата "Сигурност в .NET Framework" ще се спрем в детайли върху императивното и декларативното управление на сигурността.

Използване на автоматизирана сериализация на обекти

Сериализацията е процес на конвертиране на обект или свързан граф от обекти в поток от байтове. Десериализацията представлява обратния процес. В .NET Framework сериализацията и десериализацията се извършват автоматично от CLR, но за да се укаже, че даден обект подлежи на сериализация, се използват атрибути. Ето един пример:

```
[Serializable]
public struct User
{
   private string mLogin;
   private string mPassword;
   private string mRealName;
   // ...
}
```

В примера чрез атрибута **SerializableAttribute** се указва на CLR, че структурата **User** може да се сериализира по стандартния за CLR начин.

Сериализацията в .NET Framework ще разгледаме по-детайлно в темата за сериализация на данни.

Компонентен модел на .NET

При проектирането на .NET платформата е залегнал принципът за компонентно-ориентираното програмиране. .NET Framework дефинира компонентен модел, който установява стандарти за разработване и използване на компоненти. При разработване на .NET компоненти чрез атрибути към тях може да се дефинират метаданни, които се използват от Visual Studio .NET по време на дизайн. Ето един пример, в който чрез атрибут се указва категорията, в която да се появи свойството **`BorderColor`** в панела за настройка на свойствата за даден компонент:

```
public class SomeComponent : Control
{
   // ...

   [Category("Appearance")]
   public Color BorderColor
   {
      get { … }
      set { … }
   }

   // ...
}
```

На компонентния модел на .NET Framework ще обърнем по-специално внимание в [темата за Windows Forms](#).

Създаване на уеб услуги в ASP.NET

.NET Framework има вградена поддръжка на уеб услуги. Уеб услугите служат за обмяна на информация между отдалечени приложения посредством стандартни протоколи. За управление на поведението на уеб услугите в ASP.NET се използват атрибути. Ето един пример, в който чрез атрибута **`[WebMethod]`** се указва, че даден метод е публично достъпен като част от дадена уеб услуга:

```
public class AddService : WebService
{
   [WebMethod]
   public int Add(int a, int b)
   {
      return a + b;
   }
}
```

На уеб услугите в ASP.NET ще обърнем специално внимание в [темата за уеб услуги](#).

Взаимодействие с неуправляван (Win32) код

.NET Framework може да взаимодейства с неуправляван Win32 код: да извиква Win32 функции, да използва COM компоненти и да публикува COM компоненти. За настройка на различни параметри на взаимодействието с неуправляван код се използват атрибути. Един от тях е системният атрибут **DllImport**. Нека разгледаме следния пример на декларация на външна Win32 функция:

```
[DllImport("user32.dll", EntryPoint="MessageBox")]
public static extern int ShowMessageBox(int hWnd, string text,
   string caption, int type);
```

В примерния код чрез **DllImport** се указва, че статичният метод **ShowMessageBox(…)** е външна неуправлявана функция с име **MessageBox** от Win32 библиотеката **user32.dll**.

Ще обърнем специално внимание на атрибутите за връзка с неуправляван код в темата "Взаимодействие с неуправляван код".

Синхронизация при многонишкови приложения

.NET Framework използва вътрешно някои атрибути, за да осигури синхронизация при конкурентен достъп до даден метод от няколко нишки. Ето един пример, при който чрез специален атрибут (**System.Runtime.CompilerServices.MethodImplAttribute**) се указва, че методът **SomeMethod()** може да бъде изпълняван от най-много една нишка в даден момент:

```
[MethodImplAttribute (MethodImplOptions.Synchronized)]
public void SomeMethod() { ... }
```

Дефиниране на собствени атрибути

До момента разгледахме какво представляват атрибутите и как се ползват атрибути, дефинирани стандартно в .NET Framework или дефинирани от други разработчици. Сега ще разгледаме как можем да дефинираме собствени атрибути, които да използваме за свои специфични цели: например, когато разработваме сървърни приложения или компоненти.

За да бъде създаден потребителски атрибут, той задължително трябва да наследява класа **System.Attribute** и на компилатора трябва да се укаже към какъв вид елементи от кода може да се прилага атрибутът, т.е. какви са неговите цели. Това става с помощта на мета-атрибута **AttributeUsage**.

Собствени атрибути – пример

Да разгледаме следния пример: в даден проект има изискване всеки клас да съдържа в себе си информация за своя автор. Една възможност да се

реализира това е във всеки един от класовете да се сложи коментар, подобен на този:

```
// This class is written by Person X.
```

За четящия кода ще бъде ясно кой е авторът, но няма да е възможно тази информация да се извлича по време на изпълнение на програмата, след като сорс кодът е бил вече компилиран.

За да решим този проблем, можем вместо горния коментар да ползваме специален атрибут:

```
[Author("Person X")]
```

Въпреки че има само функциите на коментар, този атрибут може да бъде извличан от метаданните програмно чрез специално създадени за това инструменти. Ако използваме подобен атрибут, ще бъде възможно, при настъпване на изключение в нашата програма, да изведем информация не само в кой метод и кой клас е настъпило то, но също и кой е авторът на кода, в който е възникнал проблемът. Това би могло да бъде полезно при неговото решаване.

Дефиниране на собствен атрибут – пример

Нека сега дефинираме нашия атрибут за автор. Както вече отбелязахме, всички атрибути са инстанции на класове, а всеки клас, дефиниращ собствен атрибут, наследява класа **System.Attribute**. В нашия случай, когато създаваме атрибут, съдържащ името на автора на класа, можем да използваме следната дефиниция:

```
public class AuthorAttribute: System.Attribute { … }
```

Както имената на вградените атрибути, така и имената на потребителските атрибути трябва да завършват с окончанието **Attribute** (по възприетата в .NET Framework конвенция за имената).

Следващото нещо, което е необходимо, за да стане нашият клас **AuthorAttribute** потребителски атрибут, е да му приложим атрибута **AttributeUsage**. Чрез него указваме кои са целите, към които може да се прилага, и дали се допуска многократно прилагане към една и съща цел.

За да е възможно подаването на параметър при създаването на атрибута, за него трябва да се дефинира и подходящ конструктор.

Следва примерна реализация на атрибута **AuthorAttribute**:

```
using System;

[AttributeUsage(AttributeTargets.Struct |
   AttributeTargets.Class | AttributeTargets.Interface)]
```

```
public class AuthorAttribute: System.Attribute
{
   private string mName;

   public AuthorAttribute(string aName)
   {
      mName = aName;
   }

   public string Name
   {
      get
      {
         return mName;
      }
   }
}
```

Както се вижда от декларацията, нашият атрибут може да се прилага само към структури, класове и интерфейси, като към дадена цел не може да се прилага повече от веднъж (това се подразбира ако не е указано друго).

Понеже нашият атрибут е клас, който има само един конструктор, единственият начин да го инстанцираме е, като извикаме този конструктор. Следователно при използване на нашия атрибут винаги трябва да подаваме позиционния параметър за име на автор.

И така, веднъж деклариран, нашият атрибут вече може да бъде прилаган като всички останали атрибути:

```
[Author("Светлин Наков")]
class CustomAttributesDemo
{
   …
}
```

В примера към класа **CustomAttributesDemo** е приложен **AuthorAttribute**, който задава автор Светлин Наков.

Ако се опитаме да приложим **AuthorAttribute** няколко пъти към една и съща цел или да го приложим без параметри, ще получим грешка по време на компилация.

До момента дефинирахме и приложихме свой собствен атрибут, но защо ни беше това? Печелим възможността да добавяме допълнителна информация към елементи от кода и да я извличаме на по-късен етап от вече компилирания код. Възниква въпросът как точно става това извличане.

Извличане на атрибути от асемблы

По време на изпълнение на програмата може да се използва следният код, за да се извлече атрибутът, приложен върху класа **CustomAttributesDemo** от горния пример:

```
string className = "CustomAttributesDemo";
Assembly ass = Assembly.GetExecutingAssembly();
Type type = ass.GetType(className);
object[] allAttributes = type.GetCustomAttributes(false);
AuthorAttribute author = allAttributes[0] as AuthorAttribute;
Console.WriteLine("Class {0} is written by {1}. ",
   className, author.Name);
```

В примера първо се взема текущото асемблы, от него се изваждат метаданните за класа **CustomAttributesDemo**, след което се извличат всички атрибути, приложени към този клас. Накрая се взема първият атрибут от списъка и се преобразува до тип **AuthorAttribute**. Както се вижда, извлеченият атрибут е най-обикновена инстанция на класа **AuthorAttribute** и тя може да се използва така, сякаш е създадена в момента, а не е извлечена от асемблито.

Горният пример използва технологията "reflection" (отражение на типовете), на която ще обърнем по-голямо внимание в съответната тема. Засега е достатъчно да знаем, че има лесен начин атрибутите да бъдат извличани по време на изпълнение на приложението.

Мета-атрибутът AttributeUsage

Вече се сблъскахме с атрибута **AttributeUsage** в предходния пример. Нека сега си изясним по-детайлно за какво служи той и кога се използва.

AttributeUsage е системен атрибут, който се прилага към декларациите на други атрибути. В този смисъл той е мета-атрибут, т.е. предоставя метаданни за атрибутите (мета-метаданни).

Когато се използва **AttributeUsage**, на конструктора се подават два аргумента. Първият от тях е набор от флагове, указващи допустимите цели, (например класове и структури), а вторият (**AllowMultiple**) е булев флаг, указващ дали е допустимо към дадена цел да се приложи повече от една инстанция на дефинирания атрибут.

Следва пример, в който се дефинира атрибут, който служи за добавяне на коментари в кода, които, за разлика от обикновените коментари, при компилация не се губят, а се запазват в компилираните асемблита:

```
[AttributeUsage(AttributeTargets.All, AllowMultiple=true)]
public class CommentAttribute: System.Attribute
{
   private string mCommentText;
```

```
   public CommentAttribute(string aCommentText)
   {
      mCommentText = aCommentText;
   }

   public string CommentText
   {
      get
      {
         return mCommentText;
      }
   }
}
```

Коментарите, дефинирани чрез горния атрибут, могат да се прикрепят към всякакви цели: асемблы, тип, метод, поле, свойство и др., като към всяка цел може да се добавя повече от един коментар. Следва пример за използването на **`CommentAttribute`**:

```
using System;

[assembly: Comment("This is a test assembly!")]

[Comment("This class is for test purposes only.")]
[Comment("(C) Svetlin Nakov, 2005. All rights reserved!")]
class TestCommentAttribute
{
   [Comment("The name of the configuration file.")]
   public static string CONFIG_FILE_NAME = "config.xml";

   [Comment("This is the program entry point.")]
   static void Main()
   {
      ...
   }
}
```

Коментарите, прикрепени към кода по този начин, могат да се извличат от компилирания код и да се използват от приложението по време на изпълнение или от различни инструменти за работа с кода, като дебъгери, оптимизатори (execution profilers) и други.

Как се съхраняват атрибутите?

Вече знаем, че атрибутите са инстанции на някакъв клас, наследник на **`System.Attribute`**. Те се съхраняват като метаданни в асемблито и могат да бъдат извличани по време на изпълнение на програмата. Сега ще разгледаме как точно се съхраняват.

Какво се случва по време на компилация?

По време на компилация приложените към дадена цел атрибути се обработват по следния начин:

1. Компилаторът намира типа, който съответства на приложения атрибут. В нашия пример с добавянето на коментари към кода на атрибута **`Comment`** съответства класа **`CommentAttribute`**.
2. Компилаторът създава инстанция на приложения атрибут. В нашия пример се инстанцира класът **`CommentAttribute`** (който е дефиниран в нашия сорс код и е компилиран преди това).
3. Компилаторът инициализира полетата на приложения атрибут чрез параметрите, подадени в конструктора му и чрез установяване на свойствата, за които е зададена стойност. В нашия случай полето **`mCommentText`** се инициализира от конструктора на **`CommentAttribute`** с подадената за него стойност.
4. Инстанцията на атрибута, която е получена, се сериализира (представя се като последователност от байтове).
5. Сериализираната инстанция се записва в таблицата с метаданните за целта, към която е приложена.

Какво се случва при извличане на атрибут?

По време на изпълнение, когато са необходими, атрибутите се десериализират от метаданните на асемблито и се предоставят на приложението. За тях се създават най-обикновени обекти от съответните им класове и състоянието им се извлича от метаданните.

Повече за сериализацията и десериализацията ще научим в темата за сериализация в .NET Framework, но за момента можем да считаме, че чрез тези техники можем да запазваме обекти от паметта във файл или друг носител и да ги възстановяваме след време обратно в паметта.

Упражнения

1. Обяснете какво представляват атрибутите в .NET Framework. Как се прилагат атрибути? Как се прилагат атрибути с параметри? Как се задава цел при прилагане на атрибут?
2. Дефинирайте собствен атрибутен тип **`VersionAttribute`**, който може да се прилага само към типове или методи и служи за задаване на версията на даден тип или метод. Версията трябва да се състои от символен низ за самата версия и незадължителен текстов коментар. Дефинирайте подходящи конструктори и свойства за класа.
3. Създайте клас **`VersionsDemo`** с няколко метода и им приложете атрибута **`VersionAttribute`** с някакви примерни версии, на места придружени от коментари.

4. Създайте малка програма, която зарежда класа **`VersionsDemo`** и отпечатва неговата версия, както и версията на всеки негов метод заедно с текстовия коментар към нея (ако има такъв). За целта използвайте методите **`GetMethods()`** и **`GetCustomAttributes()`** на класа **`System.Type`**.

Използвана литература

1. Светлин Наков, Атрибути – http://www.nakov.com/dotnet/lectures/Lecture-6-Attributes-v1.0.ppt
2. Jeffrey Richter, Applied Microsoft .NET Framework Programming, Microsoft Press, 2002, ISBN 0735614229
3. MSDN Training, Programming with the Microsoft® .NET Framework (MOC 2349B), Module 17: Attributes
4. MSDN Library – http://msdn.microsoft.com

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 8. Масиви и колекции

Необходими знания

- Базови познания по структури от данни
- Базови познания за общата система от типове в .NET (Common Type System)
- Базови познания за езика C#

Съдържание

- Масиви в .NET Framework
- Многомерни масиви. Масиви от масиви
- Типът **`System.Array`**
- Сортиране на масиви и двоично търсене
- Колекции в .NET Framework
- **`IList`**, **`ArrayList`**, **`Queue`** и **`Stack`**
- **`IDictionary`** и **`Hashtable`**. Собствени хеш-функции
- Класът **`SortedList`**

В тази тема ...

В настоящата тема ще се спрем на масивите и колекциите в .NET Framework. Ще разгледаме видовете масиви: едномерни, многомерни и масиви от масиви (т. нар. назъбени масиви), както и базовия тип за всички масиви **`System.Array`**. Ще се запознаем с начините за сортиране на масиви и търсене в тях. Ще разгледаме с колекциите и тяхната реализация в .NET Framework: класовете **`ArrayList`**, **`Queue`**, **`Stack`**, **`Hashtable`** и **`SortedList`**, както и интерфейсите, които те имплементират.

Какво е масив?

Масивите са наредени последователности от еднакви по тип елементи. Те представляват механизми, който ни позволяват да третираме тези последователности като едно цяло.

Деклариране на масиви

Масиви в C# декларираме по следния начин:

```
int[] myArray;
```

В случая сме декларирали масив с име **myArray** от целочислен тип (**System.Int32**). В началото **myArray** има стойност **null**, тъй като не е заделена памет за елементите на масива.

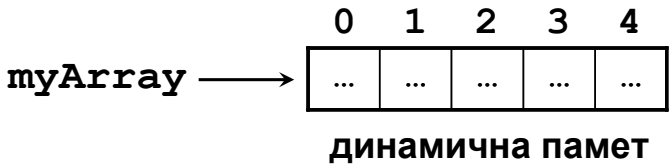
Заделяне на масиви

Със следния код заделяме (алокираме) масив в C#:

```
myArray = new int[5];
```

В примера се заделя масив с размер 5 елемента от целочислен тип.

При заделяне на масив CLR автоматично инициализира всички негови елементи с неутрална стойност (**0** или **null**). В нашия пример всеки от тези 5 елемента е със стойност **0**, за разлика от други неуправлявани среди, където стойностите ще са произволни (C, C++). Адресът на блока памет, заделен за този масив се записва в променливата **myArray**.



Масивите в .NET Framework

Всички масиви в .NET Framework наследяват типа **System.Array**, което означава, че те винаги са референтни типове и се разполагат в блокове от динамичната памет (т. нар. managed heap).

От своят страна типът **System.Array** имплементира следните интерфейси: **ICloneable**, **IList**, **IEnumerable** и **ICollection**, които позволят масивите да се използват лесно в различни ситуации. Ще разгледаме тези интерфейси малко по-късно в настоящата тема.

Тъй като са референтни типове масивите винаги се предават по референция (т.е. по адрес, а не по стойност). Ако искаме да подадем даден масив

като параметър, но да защитим от промяна стойностите на неговите елементи, трябва да подадем негово копие. Копие на масив можем да направим чрез статичния метод **`Array.Copy(…)`**. Обърнете внимание, че този статичен метод прави плитки копия на елементите на масива.

Елементи на масивите

Достъпът до елементите на масивите е пряк, по индекс (пореден номер на елемента). Масивите обикновено са нулево-базирани, т.е. номерацията на елементите започва от 0. В .NET Framework обаче могат да се създадат и масиви с ненулева долна граница. Елементите на масивите са достъпни както за четене така и за писане.

Достъпът до елементите на масивите е проверен, т.е не се допуска излизане извън границите и размерностите на масив и при всеки достъп CLR проверява дали индексът е валиден и ако не е, се подава изключение **`System.IndexOutOfRangeException`**. Естествено тази проверка си има и своята цена и това е производителността. CLR обаче ни предоставя възможността да я изключим, като използваме ключовата дума **`unsafe`**. Тя се използва винаги, когато искаме да извършваме операции свързани с указатели. С **`unsafe`** трябва да обозначим метода, който ще извършва тези операции. Ето пример:

```
unsafe static void FastArrayAccess(int[] myArray)
{
   // Acess the array elements here with no checks
}
```

При компилация на код, който използва **`unsafe`** трябва да укажем опцията на компилатора, че кодът има **`unsafe`** блокове. Ето как става това:

```
csc.exe /unsafe UnsafeArrayAccess.cs
```

Ако за нашето приложение бързодействието е от най-голямо значение можем да използваме **`unsafe`** код. Не трябва да забравяме обаче, че кодът, който пишем, вече няма да е управляван (managed) и CLR няма да се грижи за неговата обезопасеност.

Видове масиви

В .NET Framework се подържат едномерни, многомерни и масиви от масиви ("назъбени" масиви). Всеки от тези видове пази в себе си информация за броя на размерностите си (т. нар. **ранг**), както и границите на всяка от тях. CLR е оптимизиран за работа с едномерни, нулево-базирани масиви, затова се препоръчва тяхното използване, когато е възможно. Масивите могат да се инициализират при деклариране.

Масиви – пример

Ще илюстрираме работата с масиви със следния пример:

```
int[] primes = {2, 3, 5, 7, 11, 13, 17, 19};
foreach (int p in primes)
{
   Console.Write("{0} ", p);
}
Console.WriteLine();
// Output: 2 3 5 7 11 13 17 19

for (int i = 0; i < primes.Length; i++)
{
   primes[i] = primes[i] * primes[i];
}

foreach (int p in primes)
{
   Console.Write("{0} ",p);
}

Console.WriteLine();
// Output: 4 9 25 49 121 169 289 361
```

В горния пример първоначално създаваме едномерен масив от тип **System.Int32**, инициализираме го с първите 8 прости числа, след което го извеждаме в конзолата.

След това на всеки елемент му се присвоява за стойност неговия квадрат. Забележете, че се използва свойството **Length**, което връща броя на елементите на масива. В .NET Framework всеки масив знае своята дължина.

Най-накрая новополучените стойности отново извеждаме на екрана чрез цикъл, реализиран с конструкцията **foreach**. Използването на **foreach** е възможно, защото масивите реализират интерфейса **IEnumerable**.

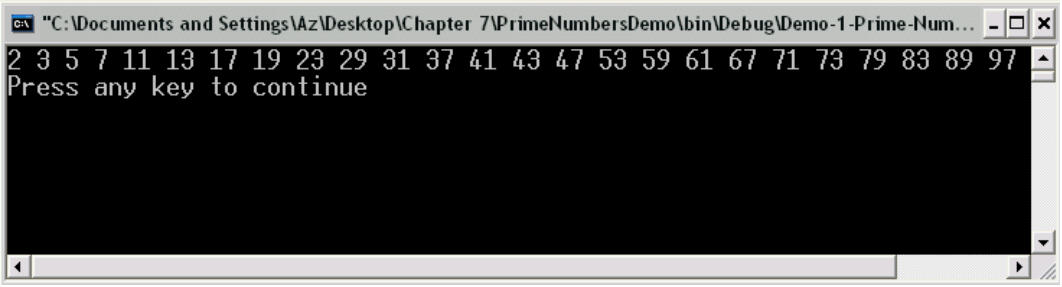
Прости числа – пример

В следващия пример ще използваме масиви за да реализираме един от най-старите алгоритми – решето на Ератостен:

```
using System;

class PrimeNumbersDemo
{
   static void Main()
   {
      const int COUNT = 100;
```

```
        bool[] prime = new bool[COUNT+1]; // array [0..100]
        for (int i = 2; i <= COUNT; i++)
        {
          prime[i] = true;
        }

        for (int p = 2; p <= COUNT; p++)
        {
          if (prime[p])
          {
            Console.Write("{0} ", p);
            for (int i = 2*p; i <= COUNT; i += p)
            {
              prime[i] = false;
            }
          }
        }

        Console.WriteLine();
    }
}
```

Резултатът от изпълнението на програмата е следният:



Как работи примерът?

Алгоритъмът на Ератостен работи по следния начин: записваме числата от 2 до n (в нашия случай 100) в редица. Първоначално всички числа са незачертани. Намираме първото незачертано число – в началото това е 2, маркираме го като просто и зачертаваме всяко кратно на 2 число в редицата. Продължаваме по същия начин със следващото незачертано число – 3. Процесът продължава докато не остане нито едно незачертано число. Тогава всички маркирани числа са прости.

Сега да се спрем по-подробно на конкретната реализация на C#. За маркирането и зачертаването на елементите ще използваме масив от тип **`System.Boolean`**. Понеже CLR инициализира по подразбиране булевите стойности с **`false`**, първият цикъл в кода им задава стойност **`true`**, като по този начин маркираме всички числа като прости. След това започваме цикъл по дължината на масива (**`COUNT`**) и за всяка негова итерация

проверяваме дали текущото число е маркирано, ако е извеждаме го и зачертаваме неговите кратни. В крайна сметка ако елемент на масива **prime** има стойност **true**, неговият индекс е просто число.

Забележка: първият цикъл започва от 2, защото както знаем 1 не е просто число.

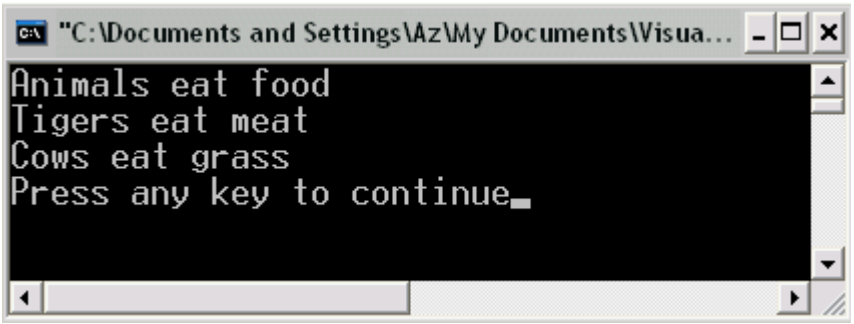
Масиви от референтни типове – пример

Със следващия пример ще илюстрираме използването на масив, чиито елементи са референтни типове, в частност – инстанции на класове дефинирани от нас.

```
using System;

class Animal
{
   public virtual void Eat()
   {
      Console.WriteLine("Animals eat food");
   }
}

class Tiger : Animal
{
   public override void Eat()
   {
      Console.WriteLine("Tigers eat meat");
   }
}

class Cow : Animal
{
   public override void Eat()
   {
      Console.WriteLine("Cows eat grass");
   }
}

class ArrayTest
{
   static void Main()
   {
      Animal[] animals = new Animal[3];
      animals[0] = new Animal();
      animals[1] = new Tiger();
      animals[2] = new Cow();

      foreach (Animal animal in animals)
      {
         animal.Eat();
```

```
        }
    }
}
```

В примера се дефинира клас **Animal**, който има виртуален метод **Eat()**. Класът **Animal** се наследява от други два класа – **Tiger** и **Cow**, които от своя страна припокриват (**override**) този виртуален метод. В **Main()** метода създаваме масив от обекти от тип **Animal**. След създаването на масива **animals**, всеки един от неговите елементи е инициализиран със стойност **null**, защото **Animal** е референтен тип и се създават само нулеви референции към него, а действителните обекти се създават на следващите 3 реда. Обърнете внимание, че в масива можем да записваме инстанции не само към **Animal**, а и към всички класове, които са негови наследници. Накрая за всеки елемент на масива извикваме метода **Eat()**. Благодарение на полиморфизма резултатът от изпълнението на програмата е следният:



Многомерни масиви

Освен вече разгледаните едномерни масиви, .NET Framework поддържа и многомерни такива (масиви с няколко размерности). Деклариране на многомерен масив е почти същото, както при едномерните, но само с една разлика – трябва да поставим запетая между размерностите му:

```
int[,] matrix = new int[3,3];
char[,,] box = new char[2,5,10];
```

Инициализиране и достъп до елементите

Ако искаме да инициализираме многомерен масив още при декларация трябва да спазим следното правило, а именно че трябва да поставим всяко измерение в отделни "кадрави скоби". Ето и пример:

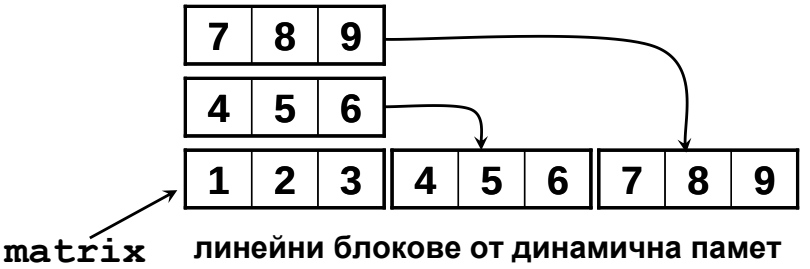
```
int[,] matrix = { {1, 2, 3} , {4, 5, 6} , {7, 8, 9} };
```

Достъпът до елементите отново е по индекс, само че в многомерния вариант отново трябва да поставим запетая между индексите на отделните размерности:

```
int elem = matrix[3,3];
box[1,2,3] = 'a';
```

Разположение в паметта

Многомерните масиви разполагат елементите си последователно – един след друг в линейни блокове от динамичната памет. Ето как би изглеждало това за вече декларирания и инициализиран с естествените числа от 1 до 9 двумерен масив **matrix**:



Многомерни масиви – пример

В следващия пример ще използваме двумерни масиви за да реализираме умножение на матрици:

```
using System;

class MatrixMultiplicationDemo
{
   static void PrintMatrix(int[,] aMatrix)
   {
      for (int row = 0; row < aMatrix.GetLength(0); row++)
      {
         for (int col = 0; col < aMatrix.GetLength(1); col++)
         {
            Console.Write("{0} ", aMatrix[row, col]);
         }
         Console.WriteLine();
      }
      Console.WriteLine();
   }

   static int[,] Mult(int[,] aMatrix1, int[,] aMatrix2)
   {
      int width1 = aMatrix1.GetLength(1);
      int height1 = aMatrix1.GetLength(0);
```

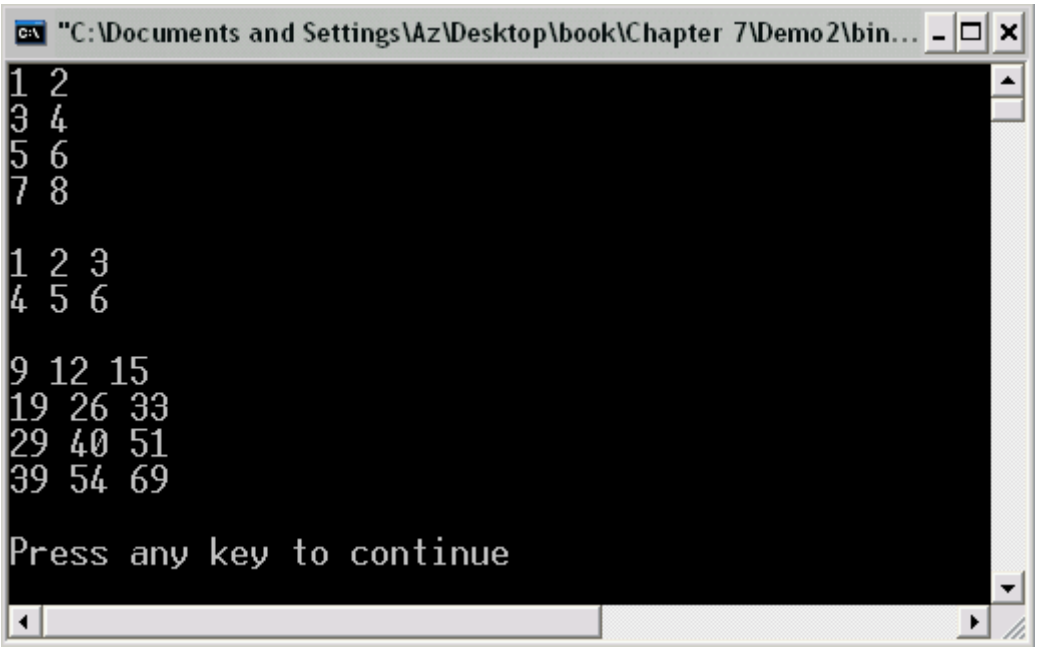
```
      int width2 = aMatrix2.GetLength(1);
      int height2 = aMatrix2.GetLength(0);

      if (width1 != height2)
      {
         throw new ArgumentException("Invalid dimensions!");
      }

      int[,] resultMatrix = new int[height1, width2];
      for (int row = 0; row < height1; row++)
      {
         for (int col = 0; col < width2; col++)
         {
            resultMatrix[row, col] = 0;
            for (int i = 0; i < width1; i++)
            {
               resultMatrix[row, col] +=
                  aMatrix1[row, i] * aMatrix2[i, col];
            }
         }
      }

      return resultMatrix;
   }

   static void Main()
   {
      int[,] m1 = new int[4,2]
      {
         {1,2},
         {3,4},
         {5,6},
         {7,8}
      };
      PrintMatrix(m1);

      int[,] m2 = new int[2,3]
      {
         {1,2,3},
         {4,5,6}
      };
      PrintMatrix(m2);

      int[,] m3 = Mult(m1, m2);
      PrintMatrix(m3);
   }
}
```

Как работи примерът?

Условието, на което трябва да отговарят две матрици за да можем да ги умножим е: броят на стълбовете на първата матрица да е равен на броя на редовете на втората матрица. Ако това не е изпълнено подаваме **`ArgumentException`**. След умножението новополучената матрица ще има следните размери: брой редове – броят на редовете на първата матрица, брой стълбове – броят на стълбовете на втората.

Самото умножение става така: всеки ред на първата матрица се умножава с всеки стълб на втората, т. е. първият елемент от реда с първия от стълба, вторият с втория и т. н. Получените произведения сумираме и това е стойността на елемента, който ще запишем в новополучената матрица на ред текущия ред от първата матрица и стълб – текущият от втората.

Ето и изхода от примера:



Масиви от масиви

В .NET Framework могат да се използват още и масиви от масиви или т. нар. **назъбени** (jagged) масиви. Може би се чудите от къде идва това име? След следващите редове ще ви се изясни.

Назъбеният масив представлява масив от масиви, т. е. всеки негов ред на практика е масив, който може да има различна дължина от останалите в назъбения масив, но не може да има различна размерност. Със следващия код декларираме масив от масиви:

```
int[][] jaggedArray;
```

Единственото по-особено е, че нямаме само една двойка скоби, както при обикновените масиви, а имаме вече две двойки такива. По следния начин заделяме назъбен масив:

```
jaggedArray = new int[2][];
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[3];
```

Възможно е и декларирането, заделянето и инициализацията на един масив от масиви да се извършва в един израз. Ето пример:

```
int[][] myJaggedArray = {
   new int[] {1,3,5,7,9},
   new int[] {17,23},
   new int[] {0,2,4,6}
};
```

Инициализиране и достъп до елементите

Достъпът до елементите на масивите, който са част от назъбения, отново е по индекс. Ето пример за достъп до елемента с индекс 3 от масива, който има индекс 0 в по-горе дефинирания назъбен масив **jaggedArray**:

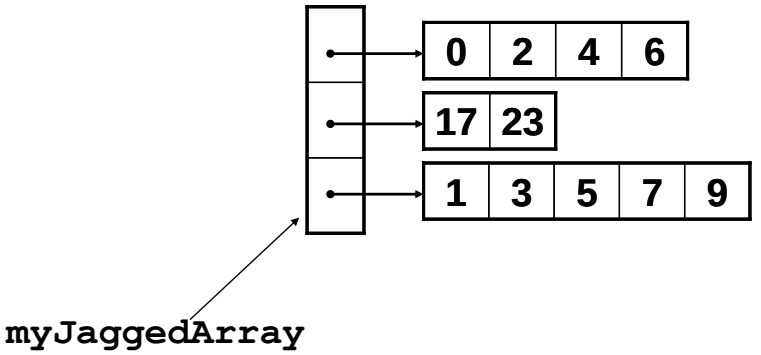
```
jaggedArray[0][3] = 12345;
```

Както споменахме, елементите на назъбения масив може и да са не само едномерни масиви, но и многомерни такива. В следващия код създаваме назъбен масив от двумерни масиви:

```
int[][,] jaggedOfMulti = new int [3][,] ;
jaggedOfMulti[0] = new int[,] { {9,27}, {10,20} };
```

Разположение в паметта

Ако все още не ви си е изяснило защо наричаме масивите от масиви – назъбени, може би тази следващата картинка ще ви помогне. На нея може да видим вече дефинирания назъбен масив **myJaggedArray** и по точно неговото разположение в паметта. Както се вижда, самият назъбен масив съдържа само референции към масивите, а не самите тях. Тъй като не знае каква ще е размерността на всеки от масивите, CLR заделя само референцията за тях. Чак след като се задели памет за някой от масивите елементи на назъбения, тогава се насочва указателя към новосъздадения блок динамична памет.



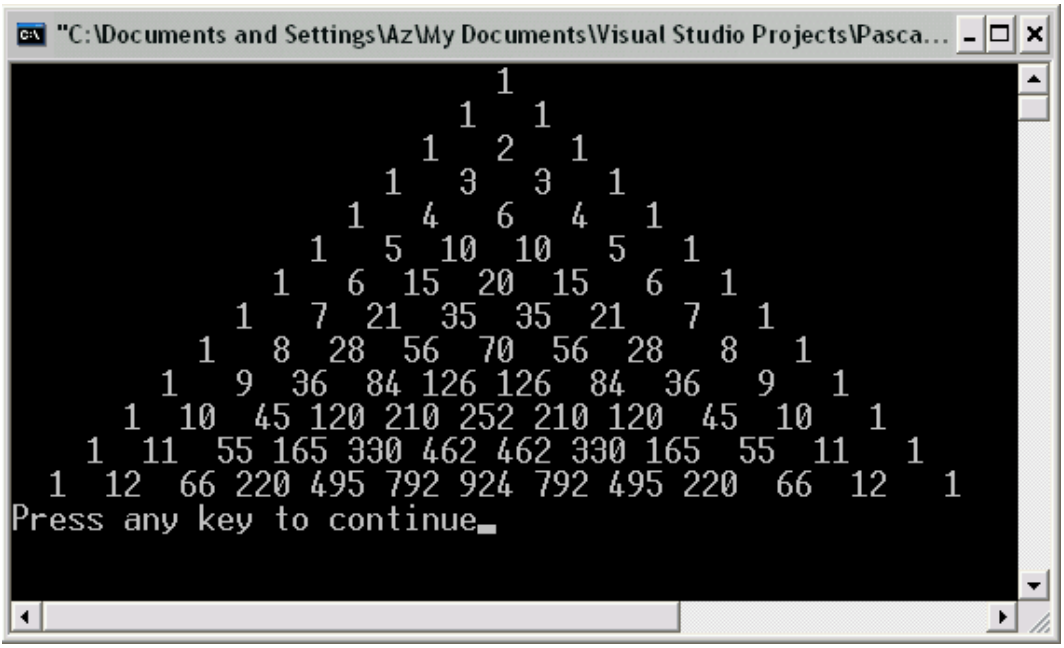
Триъгълник на Паскал – пример

В следващия пример ще използваме назъбен масив за да генерираме и визуализираме триъгълника на Паскал. Както знаем от математиката, всяко число от триъгълника се образува като се съберат горните две над него. Естествено, това не важи за първото число в триъгълника – 1. Триъгълникът на Паскал има широко приложение в комбинаториката.

```
using System;

class PascalTriangle
{
  static void Main()
  {
    const int HEIGHT = 12;

    // Allocate the array in a triangle form
    long [][] triangle = new long[HEIGHT+1][];
    for (int row = 0; row <= HEIGHT; row++)
    {
      triangle[row] = new long[row+1];
    }

    // Calculate the Pascal's triangle
    triangle[0][0] = 1;
    for (int row = 0; row < HEIGHT; row++)
    {
      for (int col = 0; col <= row; col++)
      {
        triangle[row+1][col] += triangle[row][col];
        triangle[row+1][col+1] += triangle[row][col];
      }
    }

    // Print the Pascal's triangle
    for (int row = 0; row <= HEIGHT; row++)
    {
      Console.Write("".PadLeft((HEIGHT-row)*2));
```

```
        for (int col = 0; col <= row; col++)
        {
           Console.Write("{0,3} ", triangle[row][col]);
        }
        Console.WriteLine();
      }
   }
}
```

Ето и резултата от изпълнението на програмата:



Типът System.Array

Всички масиви в .NET Framework наследяват типа **`System.Array`**. Това наследяване става неявно и се извършва от CLR. Абстрактният тип **`System.Array`** предлага няколко метода и свойства, който също се наследяват от всички масиви. В следващите таблици ще разгледаме по-важните методи и свойства на **`System.Array`**.

Свойства

Свойство	Описание
`Rank`	Връща броя на размерностите (измеренията) на масива.
`Length`	Връща цяло число от тип **`System.Int32`**, което представлява общия брой на елементите (от всич-

	ки размерности) на масива.
LongLength	Връща цяло число от тип **System.Int64**, което представлява общия брой на елементите (от всички размерности) на масива.

Забележка: Изброените свойства са достъпни само за четене.

Методи

Метод	Описание
GetLength(…)	Връща броя на елементите по дадена размерност.
GetEnumerator()	Връща имплементация на интерфейса **IEnumerator** за елементите на масива. Това дава възможност да се използва конструкцията **foreach(…)**, чрез която може да се обходят всички елементи на масива. Обхождането за многомерни масиви става отляво на дясно по размерностите, т.е. най-дясното измерение се сменя най-бързо.
Reverse(…)	Статичен метод на **System.Array**. Обръща елементите на даден едномерен масив в обратен ред. Ако масивът не е едномерен се подава **RankException**.
Clear(…)	Статичен метод на **System.Array**. Задава стойност **0** (**null** за референтни типове) на елементите в зададен интервал.
Sort(…)	Статичен метод на **System.Array**. Сортира елементите на даден масив по големина.
BinarySearch(…)	Статичен метод на **System.Array**. Търси за даден елемент в даден масив чрез метода на двоичното търсене. Методът предполага, че елементите на масива се подредени по големина предварително.
IndexOf(…)	Статичен метод на **System.Array**. Връща индекса на първото срещане на дадена стойност в даден едномерен масив. Ако елементът не се среща в масива, връща **-1**. Ако масивът не е едномерен, се подава **RankException**.
LastIndexOf(…)	Статичен метод на **System.Array**. Връща индекса на последното срещане на дадена стойност в даден едномерен масив. Ако елементът не се среща в масива връща **-1**. Ако масивът не е едномерен се подава **RankException**.
CreateInstance(…)	Статичен метод на **System.Array**. Създава дина-

	мично (по време на изпълнение) инстанция на типа **System.Array**, като може да се зададе тип на елементите, брой размерности, долна граница и брой елементи за всяка размерност.
Copy(…)	Статичен метод на **System.Array**. Копира елементите на един масив (или част от тях) в друг масив. Този метод извършва, ако е необходимо, преобразуване на типовете на масивите.

Имплементирани интерфейси

Типът **System.Array** имплементира следните интерфейси: **ICloneable**, **IList**, **IEnumerable** и **ICollection**. Тези интерфейси (методите, свойствата и индексаторите, които предлагат) улесняват използването на масивите в множество и разнообразни ситуации.

Ще разгледаме малко по-подробно всеки един от имплементираните интерфейси:

- **ICloneable** – предоставя метод **Clone()**, който се използва за клониране на масив. По подразбиране масивите се копират плитко (shallow copy). Това означава, че ако копираме масив от референтни типове, елементите на новия масив ще сочат към същото място в паметта, към което са сочили съответно елементите от стария масив.
- **IList** – предоставя директен (пряк) достъп до елементите на масива. Типът **System.Array** експлицитно (явно) имплементира всеки един от методите на **IList**.
- **IEnumerable** – предоставя метода **GetEnumerator()**, чрез който могат да се обходят всички елементи на масива (виж таблицата с методите на класа **System.Array**).
- **ICollection** – осигурява свойството **Count** (размер) и средства за синхронизация на достъпа до елементите. **ICollection** от своя страна имплементира **IEnumerable**.

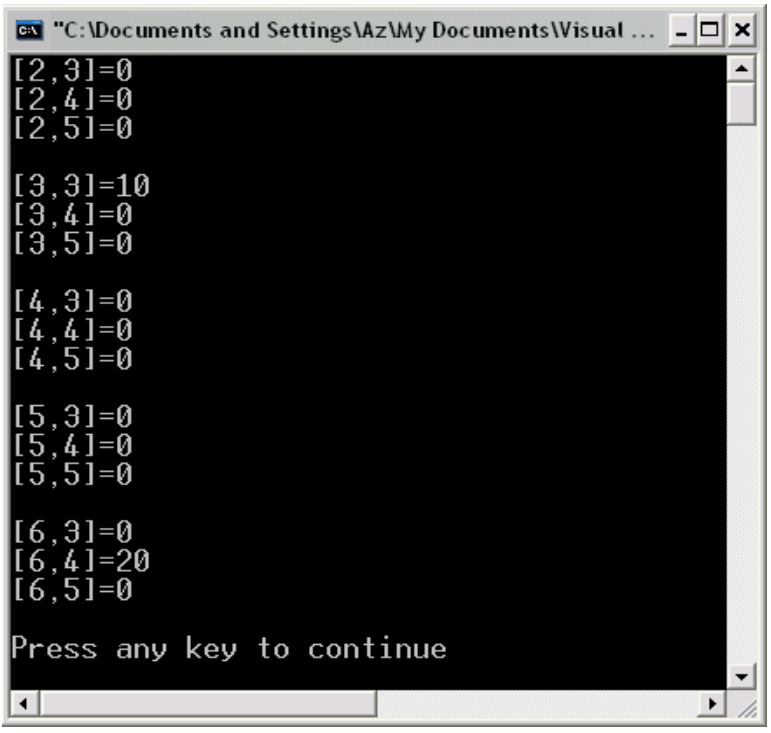
Създаване на ненулево-базиран масив – пример

В следващия пример ще илюстрираме как е възможно създаването на масив в .NET Framework, чиято долна граница не е **0**:

```
using System;

class NonZeroBasedArray
{
   static void Main()
   {
      int[] lowerBounds = { 2, 3 };
      int[] lengths = { 5, 3 };
```

```
      int[,] myArray = (int[,]) Array.CreateInstance(
        typeof(int), lengths, lowerBounds);

      myArray[3,3] = 10;
      myArray[6,4] = 20;
      // myArray[0,0] = 40; will throw IndexOutOfRangeException

      for( int i=myArray.GetLowerBound(0);
           i<=myArray.GetUpperBound(0); i++ )
      {
        for( int j=myArray.GetLowerBound(1);
             j<=myArray.GetUpperBound(1); j++ )
        {
          Console.WriteLine("[{0},{1}]={2}", i, j, myArray[i,j]);
        }
        Console.WriteLine();
      }
    }
}
```

Ето резултатът от изпълнението на програмата:



В примера използваме вече разгледания статичен метод **`Array.CreateInstance(…)`**, който приема като параметри типа на масива, който се създава, както и два масива от тип **`System.Int32`** – първият с дължи-

ните на всяка размерност, а вторият – с долните граници за всяка от размерностите. Използваме още и методите **GetLowerBound(…)** и **GetUpperBound(…)**, които връщат съответно долната и горната граница за дадена размерност.

За удобство върнатият от метода **CreateInstance(…)** обект може да се преобразува към очаквания тип. В горния пример това е типът **int[,]**.

Странно, но факт е, че в .NET Framework това преобразуване не може да стане, ако очакваният масив е едномерен и първият елемент на масива не е 0 (CLR подава **InvalidCastException**). Ето един такъв пример:

```
int[] lowerBounds = { 2 };
int[] lengths = { 5 };

int[] arr = (int[]) Array.CreateInstance(
   typeof(int), lengths, lowerBounds);
// System.InvalidCastException is thrown
```

Ако искаме да ползваме едномерни масиви, трябва да ползваме методите на **System.Array**: **GetValue(…)** и **SetValue(…)**, които ни дават достъп до елементите на масива. Следният код демонстрира това:

```
int [] lengths = {6};
int [] bounds = {5};

Array array = Array.CreateInstance(
   typeof(double), lengths, bounds);

for (int i=array.GetLowerBound(0); i<=array.GetUpperBound(0);
   i++)
{
   array.SetValue(7, i);
}
```

Сортиране на масиви

За да сортираме елементите на даден масив в .NET Framework използваме статичния метод **Sort(…)** на типа **System.Array**. Алгоритъмът, чрез който се извършва сортирането, е бързото сортиране на Хоор (QuickSort), който има сложност Θ(n log(n)) в средния случай.

Забележка: QuickSort не е "устойчив" алгоритъм за сортиране, т.е. ако два елемента са равни, не е сигурно дали тяхната подредба ще се запази. Методът **Sort(…)** има няколко предефиниции. Ще разгледаме по-важните от тях:

- **Sort(Array)** – сортира елементите на зададения едномерен масив, като очаква те да имплементират интерфейса **IComparable**. Той е имплементиран от много стандартни типове – **Int32**, **Float**, **Double**,

Decimal, **String**, **DateTime** и др. Ако сме си дефинирали наш тип (клас) и искаме да сортираме масив от такива елементи, трябва имплементираме **IComparable** и по-точно неговия виртуален метод **CompareTo(…)**.

- **Sort(Array, IComparer)** – сортира елементите на дадения едномерен масив по зададена схема за сравнение (имплементирана в интерфейса **IComparer**).
- **Sort(Array, index, length)** – сортира част от елементите на масива – в интервала [**index**, **index + length**).
- **Sort(Array, Array)** – сортира елементите на двата масива като използва елементите на първия масив като ключове, по които да сортира масивите.

Сортиране на масиви – пример

Със следващия пример ще покажем колко е лесно сортирането на даден масив в .NET Framework:

```
static void Main()
{
  String[] beers = {"Загорка", "Ариана", "Шуменско", "Астика",
    "Каменица", "Болярка", "Амстел"};

  Console.WriteLine("Unsorted: {0}",
    String.Join(", ", beers));
  // Result: Unsorted: Загорка, Ариана, Шуменско,
  // Астика, Каменица, Болярка, Амстел

  // Elements of the array beers are of type String,
  // so they implement IComparable
  Array.Sort(beers);

  Console.WriteLine("Sorted: {0}",
    String.Join(", ", beers));
  // Result: Sorted: Амстел, Ариана, Астика,
  // Болярка, Загорка, Каменица, Шуменско
}
```

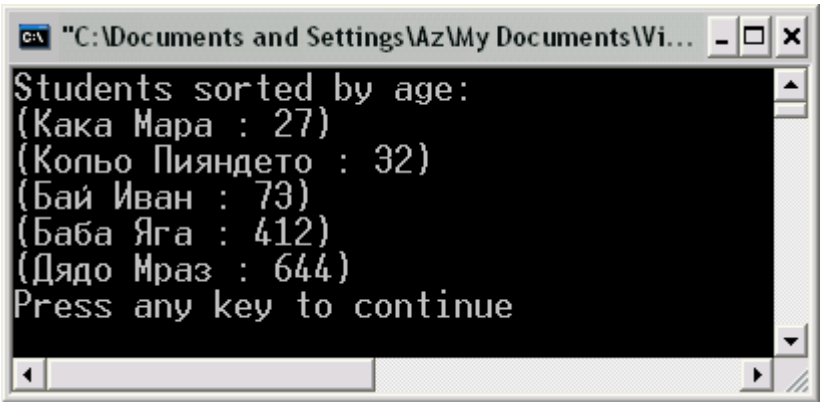
Сортиране с IComparer – пример

В следващия пример ще покажем как можем да сортираме масиви като подаваме на метода **Array.Sort(…)** като параметър тип, който имплементира интерфейса **IComparer**. За целта сме дефинирали клас **Student** и клас **StudentAgeComparer**, който имплементира **IComparer** и в метода си **Compare(…)** сравнява студентите по техните години. Забележете, че ако някой от двата обекта, които се подават като параметри на **Compare(…)**, не е **Student**, подаваме **ArgumentException**, защото няма как да ги сравним.

```
using System;
using System.Collections;

class Student
{
   internal string mName;
   internal int mAge;

   public Student(string aName, int aAge)
   {
      mName = aName;
      mAge = aAge;
   }

   public override string ToString()
   {
      return String.Format("({0} : {1})", mName, mAge);
   }
}

class StudentAgeComparer : IComparer
{
   public int Compare(object aEl1, object aEl2)
   {
      Student student1 = aEl1 as Student;
      if (student1 == null)
      {
         throw new ArgumentException(
            "Argument 1 is not Student or is null");
      }
      Student student2 = aEl2 as Student;
      if (student2 == null)
      {
         throw new ArgumentException(
            "Argument 2 is not Student or is null");
      }
      return student1.mAge.CompareTo(student2.mAge);
   }
}

class CompareStudentsDemo
{
   static void Main()
   {
      Student[] students =
      {
         new Student("Бай Иван", 73),
         new Student("Дядо Мраз", 644),
         new Student("Баба Яга", 412),
         new Student("Кака Мара", 27),
```

```
            new Student("Кольо Пияндето", 32)
        };

        Array.Sort(students, new StudentAgeComparer());

        Console.WriteLine("Students sorted by age:");
        foreach (Student student in students)
        {
            Console.WriteLine(student);
        }
    }
}
```

Ето и изходът от примера:



Двоично търсене

Когато искаме многократно да търсим различни елементи в даден масив, е по-добре първо да го сортираме и после да използваме метода на двоичното търсене. Това е бърз метод за претърсване на вече сортиран масив. Сложността, с която претърсва масив от n елемента, е Θ(log(n)).

В .NET Framework двоичното търсене е реализирано в статичния метод **`Array.BinarySearch(Array, object)`**. Ако елементът бъде намерен, методът връща неговия индекс, а в противен случай връща отрицателно число, което е побитово отрицание на индекса на първия елемент, който е по-голям от търсения или побитово отрицание на индекса на последния елемент + 1, ако търсената стойност е по-голяма от всички елементи в масива. Методът **`Array.BinarySearch(…)`** има същите изисквания като **`Array.Sort(…)`** – или елементите на масива трябва да имплементират **`IComparable`** или трябва да се подаде инстанция на **`IComparer`**.

Двоично търсене – пример

Със следващия пример ще покажем използването на статичния метод **`Array.BinarySearch(…)`** за да претърсим масива за дадени елементи. Ще

използваме и оператора за побитово отрицание ~ за да видим на коя позиция е следващия по-големина елемент, ако елементът, който търсим не бъде намерен в масива.

```
static void Main()
{
   String[] beers = {"Загорка", "Ариана", "Шуменско", "Астика",
      "Каменица", "Болярка", "Амстел"};
   Array.Sort(beers);

   string target = "Астика";
   int index = Array.BinarySearch(beers, target);
   Console.WriteLine("{0} is found at index {1}.",
      target, index);
   // Result: Астика is found at index 2.
   target = "Мастика";
   index = Array.BinarySearch(beers, target);
   Console.WriteLine("{0} is not found. The next larger element
      is at index {1}.", target, ~index);
   // Result: Мастика is not found. The next larger element is
      at index 6.
}
```

Съвети за работа с масиви

Ето няколко съвета, които ще ви помогнат да се справите с по-често срещаните грешки, възникващи при работа с масиви:

- Когато даден метод връща масив и трябва да върнете празен масив, връщайте масив с 0 елемента, а не **`null`**. Когато използвате този метод ще очаквате да се върне масив, макар и с 0 елемента, а не стойност **`null`**, ако няма елементи.
- Не забравяйте, че масивите се предават по референция и затова ако искате да сте сигурни, че даден метод няма да промени даден масив, подавайте на метода копие от масива.
- Методът **`Clone()`** връща плитко копие на масива. Ако елементите на масива са референтни типове, трябва да реализирате собствено дълбоко клониране.
- При копиране на даден масив в друг използвайте метода **`Copy(…)`**, а не **`Clone()`**. Статичният метод **`Copy(…)`** е по-удачен, защото когато е необходимо извършва съответните преобразувания (опаковане и разопаковане) на типовете на елементите на масивите. Ако масивът е от референтни типове, имайте предвид, че **`Copy(…)`** създава плитко копие на референциите.
- Не използвайте метода **`Array.BinarySearch(…)`** върху масив, който не е сортиран. Резултатите, които ще върне методът, ако масивът не е сортиран, са трудно предвидими.

- Когато е възможно използвайте само едномерни, нулево-базирани масиви. CLR е оптимизиран за работа с тях и производителността на вашето приложение ще е много по-добра.

Какво е колекция?

Колекции наричаме класовете, които съдържат в себе си съвкупност от елементи, най-често от един и същи тип. Колекциите са известни още като "контейнер класове" или "контейнери".

Колекциите могат да бъдат с фиксиран размер (такива са например масивите) или с променлив размер (такива са например свързаните списъци). Те могат да бъдат само за четене или да позволяват и промени.

Колекциите са абстрактни типове данни и могат да бъдат имплементирани по различен начин, например: чрез масив, чрез свързан списък, чрез различни видове дървета, чрез пирамида, чрез хеш-таблица и т. н.

Колекциите от обекти са важен елемент в обектно-ориентирания дизайн. Те позволяват даден клас да съдържа множество обекти от даден друг клас. По този начин могат да бъдат моделирани различни взаимоотношения между класовете – асоциации, агрегации, композиции и др.

Колекциите в .NET Framework

В .NET Framework класовете, имплементиращи колекции, се намират в пространството от имена **`System.Collections`**. Пример за колекция от **`System.Collections`** е типът **`Hashtable`**. Той имплементира хеш-таблици, които се характеризират с изключително бърз достъп по ключ до съдържаните елементи. Друг пример е **`ArrayList`**, който реализира масиви с променлива дължина. Използването на тези и други типове, дефинирани в **`System.Collections`**, ни позволява да прекарваме повече време, пишейки код, реализиращ същината на нашето приложение, вместо да си губим времето в опити да реализираме често срещани структури.

Списъчни и речникови колекции

Колекциите в C# са два вида – списъчни и речникови. Списъчните се характеризират с това, че имплементират интерфейсите **`IList`** и **`ICollection`**. Такива са например **`ArrayList`**, **`Queue`**, **`Stack`**, **`BitArray`** и **`StringCollection`**. Речниковите колекции имплементират интерфейса **`IDictionary`** и по-точно представляват колекция от двойки (ключ, стойност). Примери за такива колекции са класовете **`Hashtable`**, **`SortedList`** и **`StringDictionary`**.

Колекциите са слабо типизирани

Основна характеристика на всички колекции от **`System.Collections`** (с изключение на **`BitArray`**, който съдържа булеви стойности) е, че те са

слабо типизирани – елементите им са от тип **System.Object**. Слабата типизация позволява на колекциите да съдържат фактически всякакъв тип данни, защото всеки тип данни в .NET Framework наследява **System.Object**. Ето един пример:

```
ArrayList list = new ArrayList();
list.Add("beer"); // string inherits System.Object
string s = (string) list[0];
```

За съжаление слабата типизация означава още, че трябва всеки път при достъп до елемент от колекцията да се прави преобразуване на типовете. Както е показано в горния пример за да се достъпи **string** от **ArrayList** е необходимо преобразуване.

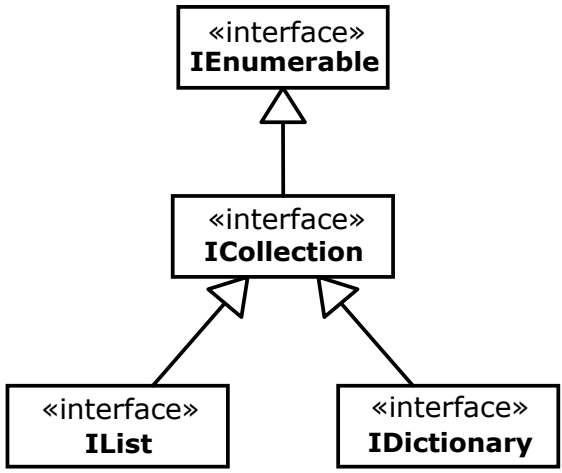
При съхранение на стойностни типове в колекции се наблюдава намалена производителност, тъй като те се преобразуват в референтни и се прави "опаковане" (boxing) и при преобразуването им обратно към стойностни, съответно – "разопаковане" (unboxing).

За да се избегне преобразуването на типове, може да се използват класовете от **System.Collections** – **CollectionBase** и **DictionaryBase** като базови за силно типизирани потребителски колекции.

В .NET Framework 2.0 ще има типизирани колекции (базирани на т. нар. generics). Те ще наподобяват така наречените шаблони (templates) в C++ и се очаква до голяма степен да решат проблема с намалената производителност и липсата на типизация.

Интерфейсите за колекции

Всички колекции в .NET Framework имплементират един или няколко от интерфейсите **IEnumerable**, **ICollection**, **IDictionary** и **IList**. На фигурата по-долу е показана клас-диаграма, изобразяваща нагледно йерархията на тези интерфейси:



Нека първо разгледаме интерфейсите **IEnumerable** и **ICollection** и обясним за какво служат. На останалите два (**IList** и **IDictionary**) ще се спрем малко по-късно.

Интерфейсът IEnumerable

Интерфейсът **IEnumerable** е базов за всички типове в .NET Framework, които поддържат операцията "обхождане на всичките им елементи". Той дефинира само един метод – методът **GetEnumerator()**, който връща итератор (инстанция на интерфейса **IEnumerator**), с който се извършва самото обхождане.

Интерфейсът **IEnumerator** дефинира методи и свойства за извличане на текущия елемент и за преминаване към следващия (ако има такъв).

Интерфейсът ICollection

Интерфейсът **ICollection** е базов за всички колекции в .NET Framework. Той разширява **IEnumerable** и добавя към него свойството **Count**, което връща общия брой елементи в дадена колекция.

Списъчни колекции

Нека първо разгледаме списъчните колекции в .NET Framework, тъй като те за по-прости от речниковите и работата с тях е по-интуитивна.

Интерфейсът IList

Всеки клас, който имплементира интерфейса **IList**, поддържа множество от стандартни операции за работа с индексиран списък: достъп до елементите по индекс, добавяне на елемент (**Add(…)**), вмъкване на елемент (**Insert(…)**), търсене на елемент (**IndexOf(…)**), изтриване по индекс или по стойност (**RemoveAt(…)**, **Remove(…)**) и др.

Класът ArrayList

Класът **ArrayList** имплементира интерфейса **IList** чрез масив, чийто размер се променя динамично при нужда. Всяка инстанция на този клас предварително заделя буферна памет (**Capacity**) за елементите, които предстои да бъдат добавени. При запълване на буферната памет се заделя нова памет, като най-често капацитетът се удвоява.

ArrayList – пример

Следният пример демонстрира работата с **ArrayList** и начина на употреба на най-честите операции, прилагани върху него:

```
static void Main()
{
   ArrayList list = new ArrayList();
```

```
    for (int i = 1; i <= 10; i++)
    {
        list.Add(i); // Adds i at the end of the ArrayList
    }
    list.Insert(3, 123); // Inserts 123 element number 3
    list.RemoveAt(7); // Removes element with the index 7
    list.Remove(2); // Removes element with value 2
    list[1] = 500; // Changes element with index 1
    list.Sort(); // Sorting in ascending order
    int[] arr = (int[])list.ToArray(typeof(int));
    foreach(int i in arr)
    {
        Console.Write("{0} ",i);
    }
    Console.WriteLine();
    // Result: 1 4 5 6 8 9 10 123 500
}
```

В горния пример използвахме някои от методите на класа **`ArrayList`**. Използвахме конструктора по подразбиране за да създадем списъка **`list`**.

Капацитет на ArrayList

Ако предварително се знае приблизителният брой елементи, които ще се добавят в **`ArrayList`**, е препоръчително да се укаже този брой още при създаването му. Това намалява броя на операциите по промяна на размера, което значително подобрява производителността.

Преоразмеряването е бавна и неефективна операция, защото е свързана със заделяне на нова памет за елементите на масива и преместване на всички стойности от старата в новата памет.

Следващият код създава **`ArrayList`** съдържащ 100 000 цели числа:

```
ArrayList list = new ArrayList();
for (int i=0; i<100000; i++)
{
    list.Add(i);
}
```

За сравнение следния код прави същото, но за два пъти по кратко време:

```
ArrayList list = new ArrayList(100000);
for (int i=0; i<100000; i++)
{
    list.Add(i);
}
```

Премахване и добавяне на елементи от ArrayList

За да се премахнат елементи от **ArrayList** се използват методите **Remove(…)**, **RemoveAt(…)**, **RemoveRange(…)** или **Clear()**. Когато се премахнат елементи, останалите, които са с по-големи индекси, автоматично се изместват наляво за да заемат мястото на вече премахнатите. Примерно, ако премахнете елемент с индекс 5, елементът с индекс 6 става с индекс 5, елементът с индекс 7 става с индекс 6 и т.н.

За да се вмъкнат елементи в **ArrayList** се използват методите **Insert(…)** и **InsertRange(…)**, като им се подава като аргумент позицията, на която се вмъкват елементите. Тъй като при вмъкване и изтриване на елемент от **ArrayList** се налага преместване на елементите от списъка и евентуално заделяне на памет, тези операции са бавни (работят с линейна сложност).

Както вече отбелязахме, всяка инстанция на **ArrayList** автоматично заделя памет за новите елементи. Тя обаче не се грижи за автоматичното освобождаване на тази памет при премахване на елементи от колекцията. За да намалите размера на **ArrayList** до настоящия брой елементи се използва метода **TrimToSize()**. Следващият пример добавя 1000 цели числа в **ArrayList**, изтрива първите 500 и след това намалява размера на колекцията до размера, нужен за останалите 500:

```
ArrayList list = new ArrayList (1000);
for (int i=0; i<1000; i++) // Add items
{
  list.Add(i);
}

list.RemoveRange(0, 500); // Remove first 500 items

list.TrimToSize(); // Resize the capacity of ArrayList
```

Други списъчни колекции

Освен **ArrayList** в .NET Framework стандартно са имплементирани и някои други списъчни колекции, като опашки и стекове.

Опашка – Queue

В .NET Framework освен **ArrayList** са дефинирани и други списъчни структури. Една от тях е така наречената опашка – **Queue**. Опашката представлява колекция с поведение от вида "first-in, first-out (FIFO)" и е реализирана чрез цикличен масив. Класическа аналогия за тази структура е опашката за билети. Този, който първи се е наредил на опашката, ще може първи да си купи билет.

Опашката е структура, която може да се използва при управление на достъпа до ограничени ресурси. Примерно, ако трябва да изпратите съобщения през даден ресурс който може да обработва само по едно

наведнъж, е подходящо да се използва опашка, която да съхранява съобщенията, които чакат да бъдат обработени.

Характерни за класа **Queue** са двата метода **Enqueue(…)** и **Dequeue(…)**, служещи съответно за добавяне и изваждане на елемент от опашката. **Enqueue(…)** добавя елемент в края на опашката, а **Dequeue(…)** изважда елемент от началото ѝ.

Стек – Stack

Подобно на опашката стекът е друга списъчна структура от пространството **System.Collections**. За разлика обаче от опашката, стекът представлява структура с поведение от вида "last-in, first-out (LIFO)", която се реализира чрез масив. Той работи на принципа "който е влязъл последен в стека, стои най-отгоре" – точно като колона от чинии, поставени една върху друга.

Основните методи за добавяне и премахване на елемент от стека са **Push(…)** и **Pop()**. **Push(…)** добавя елемент към върха на стека, а **Pop()** връща елемента от върха на стека, като го премахва. Класът **Stack** съдържа и още метода **Peek()**, който връща елемента от върха на стека, но без да го премахва.

Други стандартни списъчни структури са **StringCollection** и **BitArray**. **StringCollection** е аналог на **ArrayList**, но за **string** обекти. **BitArray**, както подсказва и името му, реализира масив от булеви стойности, като съхранява всяка една от тях в 1 бит.

Queue и Stack – примери

Ето и два кратки примера за използването на стек и опашка:

```
Queue queue = new Queue();
queue.Enqueue("1. IBM");
queue.Enqueue("2. HP");
queue.Enqueue("3. Microsoft");

while (queue.Count > 0)
{
   string computer = (string) queue.Dequeue();
   Console.Write("{0} ", computer);
}
Console.WriteLine();
// Result: 1. IBM 2. HP 3. Microsoft
```

В следващия пример на мястото на опашка се ползва стек:

```
Stack stack = new Stack();
stack.Push("1. IBM");
stack.Push("2. HP");
stack.Push("3. Microsoft");
```

```
while (stack.Count > 0)
{
   string computer = (string) stack.Pop();
   Console.Write("{0} ", computer);
}
Console.WriteLine();
// Result: 3. Microsoft 2. HP 1. IBM
```

Речникови колекции

Нека сега разгледаме и по-сложната част от средствата за работа с колекции в .NET Framework – речниковите колекции.

Интерфейсът IDictionary

Интерфейсът **IDictionary** е базов за речниковите колекции. Всеки техен елемент представлява двойка от тип ключ-стойност, която се съхранява в обект от тип **DictionaryEntry**. Ключът на всяка двойка трябва да е уникален и различен от **null**, а стойността, асоциирана с този ключ, може да е какъвто и да е обект, включително **null**. Интерфейсът **IDictionary** позволява съдържаните в колекцията ключове да се изброяват, но не ги сортира по какъвто и да е признак.

IDictionary поддържа операциите добавяне на нова двойка ключ-стойност (**Add(…)**), търсене на стойност по ключ (индексатор), премахване на двойка по ключ (**Remove(…)**), извличане на всички ключове (**Keys**), извличане на всички стойности (**Values**).

Имплементациите на **IDictionary** биват няколко вида: само за четене (read-only), с фиксиран размер и с променлив размер. При колекциите, които са само за четене, не се позволява промяна на елементите им. При имплементация с фиксиран размер не се позволява добавяне и премахване на елементи, но е позволена промяната на вече съществуващи елементи. При имплементация със променлив размер е позволено добавяне, премахване и промяна на елементи.

Класът Hashtable

Hashtable представлява имплементация на структурата от данни "хеш таблица" – речникова колекция, елементите на която се разполагат в специално заделена памет в зависимост от хеш кода на ключа на всяка от тях. Имплементацията на класа е направена така, че да позволява добавянето на елемент и търсенето по ключ да стават с константна сложност в средния случай. Тъй като класът имплементира **IDictionary**, това означава, че всеки ключ трябва да е уникален и различен от **null**.

Обектите, които се използват за ключове в хеш-таблица, трябва да имплементират или наследяват методите **GetHashCode()** и **Equals(…)**. Структу-

рата от данни "хеш-таблица" не може да работи без функция за пресмятане на хеш-код за съхраняваните ключове и без функция за сравнение на ключове. При това ключовете, които се считат за еднакви, задължително трябва да имат еднакъв хеш-код.

Тъй като всеки клас наследява **System.Object**, то той автоматично наследява и предефинирана имплементация на **Equals(…)**. За съжаление, в общия случай тази имплементация се реализира чрез сравнение за съвпадение на референциите на двата обекта. Това генерално погледнато е грешен начин за сравнение и затова се налага да се имплементират специфични реализации за създадените от нас класове. Имплементацията на **Equals(…)** трябва да връща винаги един и същ резултат, когато се вика с едни и същи параметри.

Докато има записани някакви елементи в хеш-таблицата, ключовете им не трябва да се променят! В противен случай търсенето по ключ може да не работи правилно. Този проблем може да се получи, ако се използва за ключ референтен тип.

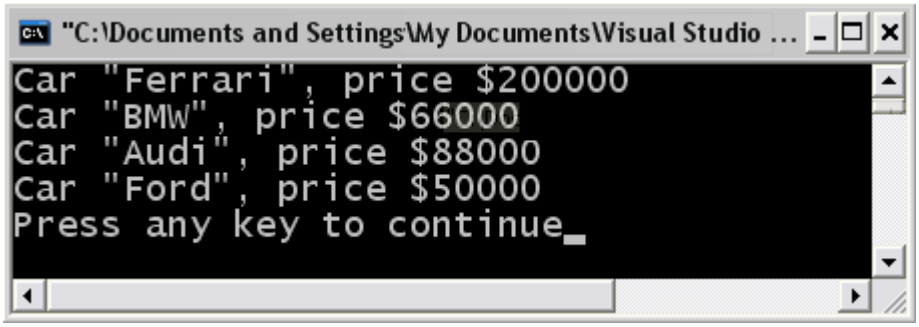
Класът **Hashtable** имплементира интерфейса **IDictionary**, а той от своя страна наследява **IEnumerable**. Това ни позволява свободата да използваме оператора **foreach** за да обхождаме елементите на хеш-таблици.

Hashtable – пример

Ето един пример, който демонстрира работата с класа **Hashtable** и показва как се използват основните операции, свързани с него:

```
static void Main()
{
   Hashtable priceTable = new Hashtable();
   priceTable.Add("BMW", 66000);
   priceTable.Add("Ferrari", 200000);
   priceTable.Add("Ford", 50000);
   priceTable.Add("Audi", 80000);
   Console.WriteLine("Car \"{0}\", price ${1}",
      "Ferrari", priceTable["Ferrari"]);
   priceTable.Remove("Ferrari");
   priceTable["Audi"] = 88000;
   foreach(DictionaryEntry carPrice in priceTable)
   {
      Console.WriteLine("Car \"{0}\", price ${1}",
         carPrice.Key, carPrice.Value);
   }
}
```

В примера се използва хеш-таблица за съхранение на съответствия между модели леки автомобили и техните цени. След изпълнението му се получава следният резултат:



Примерът демонстрира добавяне, извличане, изтриване и промяна на елементи в хеш-таблица, както и обхождане на всички елементи с **`foreach`**.

Производителност на Hashtable

Два фактора оказват влияние върху производителността на хеш-таблиците: техният размер и уникалността на хеш-кодовете, които се съпоставят на ключовете.

Влияние на колизиите върху производителността

Колизия се получава, когато на два различни ключа на елементи от хеш-таблицата се съпостави един и същ хеш-код. **`Hashtable`** използва алгоритъм с двойно хеширане за да намали негативния ефект на колизиите върху производителността, но най-добра производителност се постига когато няма никакви колизии.

Размерът на една хеш-таблица автоматично нараства при запълване до определен процент в резултат от добавяне на нови елементи, с което свежда до минимум вероятността да се получат колизии.

Влияние на размера на хеш-таблицата върху производителността

Операциите, предизвикващи увеличаване на размера на една хеш-таблица, са скъпи, защото предизвикват заделяне на нова памет, ново пресмятане на индексите за елементите и копиране на всеки елемент на нова позиция в таблицата. По подразбиране хеш-таблицата се конструира с размер 0. Това означава, че са необходими много операции по заделяне на памет докато се достигне подходящ размер.

Ако предварително знаете приблизително колко елемента ще добавите в хеш-таблицата, задавайте началния й размер като параметър на конструктора. Ето пример за инициализиране на хеш-таблица, която е оптимизирана за 1000 елемента:

```
Hashtable table = new Hashtable(1000);
```

Инициализирайки по такъв начин размера на хеш-таблицата, ние не указваме влияние върху производителността на търсещите операции, но можем да подобрим бързодействието на добавянето на нов елемент до няколко пъти.

Фактор на нарастване

Когато една хеш-таблица в .NET Framework нараства, тя винаги приема размер, който е просто число. (Причината за това се дължи на факта, че статистически е по-вероятно, ако **n** е произволно число, **n % m** да бъде уникален резултат, когато **m** е просто). По подразбиране хеш-таблицата увеличава размера си, когато броят на елементите й надвиши даден процент от размера й. Този процент може да се контролира като се променя **факторът на нарастване**. Фактор на нарастване 1.0 отговаря на 72%, 0.9 отговаря на 65% (0.9*72) и т.н. Валидните фактори на нарастване варират от 0.1 до 1.0.

Ето пример как да инициализираме хеш-таблица за 1000 елемента и да й зададем фактор на нарастване 0.8, което означава че таблицата ще нараства и ще се реиндексира когато броят на елементите й достигне приблизително 58% от размера й:

```
Hashtable table = new Hashtable (1000, 0.8f);
```

Факторът на нарастване по подразбиране е 1.0 и в повечето случаи не е нужно да се променя.

Увеличаване на уникалността на хеш-кодовете

Увеличаване до възможно най-висока степен уникалността на хеш-кодовете, генерирани от ключовете, е от голяма важност за производителността на хеш-таблиците. При по-уникални кеш-кодове се получават по-малко колизии и от там търсенето и добавянето работят по-бързо.

По подразбиране хеш-таблиците хешират ключовете, като викат техния метод **GetHashCode()**, който всеки обект е наследил от **System.Object**. Ако използваме за ключове обекти от клас, чийто метод **GetHashCode()** не генерира достатъчно уникални хеш-кодове, можем да направим едно от следните неща за да подобрим производителността:

- Да припокрием метода **GetHashCode()** в производния клас и да осигурим имплементация, която генерира възможно по-уникални хеш-кодове.
- Да създадем тип, който имплементира **IHashCodeProvider** и да подадем референция към обект от този тип на конструктора на хеш-таблицата. Това ще предизвика извикване на метода **IHashCodeProvider.GetHashCode()** за генериране на хеш-код.

Собствени хеш-функции

Когато използваме собствени класове за ключове в хеш-таблица, те трябва да припокрият методите **GetHashCode()** и **Equals(…)** наследени от **System.Object**.

Собствени хеш-функции – пример

В следващият пример ще демонстрираме как можем да използваме собствен клас за ключ в хеш таблица, като дефинираме по подходящ начин в него виртуалните методи **GetHashCode()** и **Equals(…)**:

```
class Student
{
   protected string mName;
   protected int mAge;

   public Student(string aName, int aAge)
   {
      mName = aName;
      mAge = aAge;
   }

   public override string ToString()
   {
      return String.Format("({0}, {1})", mName, mAge);
   }

   public override bool Equals(object aStudent)
   {
      if ((aStudent==null) || !(aStudent is Student))
         return false;
      Student student = (Student) aStudent;
      bool equals = (mName == student.mName) &&
         (mAge == student.mAge);
      return equals;
   }

   public override int GetHashCode()
   {
      int hashCode = mName.GetHashCode() ^ mAge;
      return hashCode;
   }
}

class CustomHashCodesDemo
{
   private static Hashtable mAddressTable;

   static void PrintAddress(Student aStudent)
   {
```

```
      if (mAddressTable.ContainsKey(aStudent))
      {
        Console.WriteLine("{0} has address: {1}.",
          aStudent, mAddressTable[aStudent]);
      }
      else
      {
        Console.Write("There is no address for {0}.", aStudent);
      }
    }

    static void Main()
    {
      Student john = new Student("Sir John", 23);
      Student ann = new Student("Lady Ann", 22);
      Student richard = new Student("King Richard", 33);

      mAddressTable = new Hashtable();
      mAddressTable.Add(john, "Nottigham Castle");
      mAddressTable.Add(ann, "London Tower");
      mAddressTable.Add(richard, "London Castle");

      PrintAddress(john);
      PrintAddress(new Student("Lady Ann", 22));
      PrintAddress(new Student("Lady Ann", 24));
    }
}
// Result:
// (Sir John, 23) has address: Nottigham Castle.
// (Lady Ann, 22) has address: London Tower.
// There is no address for (Lady Ann, 24).
```

В примера се използва класът **Student** като ключ в хеш-таблица, която съхранява съответствия между студенти и техните адреси. Ако в класа **Student** нямаме имплементация на методите **Equals(…)** и **GetHashCode()**, нашата хеш-таблица няма да работи коректно с ключове от тип **Student**.

Много често наш, собствен тип се състои от няколко различни полета, от които може да зависи хеш-кода на отделен обект от дадения тип. Един възможен начин за генериране на хеш код в такива ситуации е да се обединят хеш-кодовете на различните полета като се използва операцията ^ (изключващо или). Точно така се реализира генерирането на хеш код и в примера по-горе.

Изчисляването на хеш-кода за даден студент става като изчислим хеш-кода за неговото име и извършим операцията "изключващо или" с неговите години. По този начин се стремим да получаваме възможно по-уникални стойности за хеш-кодовете. При еднакви имена или еднакви години най-често стойностите ще са различни. Ако имената и годините на двама студента едновременно съвпадат, ще получим еднакви хеш-стой-

ности (което е едно от важните изисквания при имплементацията на метода **GetHashCode()**).

Equals(…) е имплементиран така, че да връща винаги **false** за различни инстанции на обекти от тип **Student** и **true** за еднакви.

Класът SortedList

Речниковите колекции могат да бъдат реализирани не само в хеш-таблици, но и по други начин. Класът **SortedList**, например, представлява имплементация на интерфейса **IDictionary**, която прилича както на хеш-таблица, така и на масив. Тази колекция съхранява двойки елементи ключ-стойност, сортирани по ключ, като позволява индексиран достъп. Тъй като е нужна непрекъсната поддръжка на сортирана последователност, **SortedList** работи доста бавно (повечето операции имат линейна сложност).

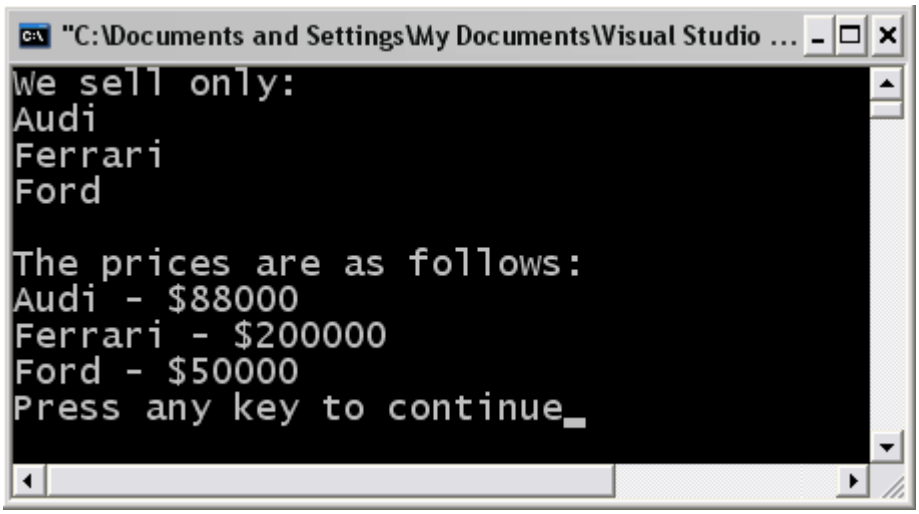
SortedList – пример

Ето един пример, който илюстрира работата със **SortedList**:

```
SortedList sl = new SortedList();
sl.Add("BMW", 66000);
sl.Add("Ferrari", 200000);
sl.Add("Audi", 80000);
sl.Add("Ford", 50000);

sl.Remove("BMW");
sl["Audi"] = 88000;

Console.WriteLine("We sell only:");
foreach (string car in sl.GetKeyList())
{
  Console.WriteLine("{0} ", car);
}

Console.WriteLine("\nThe prices are as follows:");
for (int i = 0; i < sl.Count; i++)
{
  Console.WriteLine("{0} - ${1}",
  sl.GetKey(i), sl.GetByIndex(i));
}
```

В него класът **SortedList** е използван за да съхранява съответствия между модели леки автомобили и техните цени. Понеже за ключ се използват моделите на автомобилите, след добавянето на няколко автомобила в сортирания списък, те могат да бъдат извлечени след това в азбучен ред чрез просто обхождане.

След изпълнение примерът извежда на конзолата следното:



Силно типизирани колекции

.NET Framework поддържа и силно типизирани колекции. Пространството от имена, което съдържа класовете за работа с тях е **System.Collections.Specialized**. Пример за силно типизирана колекция е **StringDictionary**. Това е клас, който работи точно като **Hashtable**, но използва само **string** за ключове и стойности. При използване на този клас няма нужда от преобразуване на типа към **string** при извличане на стойност от хеш-таблицата:

```
StringDictionary addresses = new StringDictionary();
addresses["доктор Иванов"] = "с. Гинци, на площада";
addresses["бат Сали"] = "гр. София, кв. Факултета";
Console.WriteLine("{0} живее в {1}",
  "доктор Иванов", addresses["доктор Иванов"]);
```

Специални колекции

.NET Framework поддържа и някои специални колекции, които приличат на вече разгледаните, но имат малко по-различно предназначение. Пример за специален тип колекция е хеш-таблица от низове, която не различава главни от малки букви в ключа на елементите. В .NET Framework такава колекция можем да получим чрез метода **CreateCaseInsensitiveHashtable()** на класа **CollectionsUtil**:

```
Hashtable addresses =
  CollectionsUtil.CreateCaseInsensitiveHashtable();
addresses["бай Иван"] = "с. Мугла";
Console.WriteLine("{0} живее в {1}",
  "БАЙ ИВАН", addresses["БАЙ ИВАН"]);
```

Упражнения

1. Напишете програма, която прочита от конзолата N цели числа, записва ги в масив и отпечатва тяхната сума и средното им аритметично.
2. Напишете програма, която прочита от конзолата масив от числа и намира в него най-дългата поредица от числа, такива че всяко следващо да е по-голямо от предходното.
3. Напишете програма, която прочита от конзолата масив от N числа и намира в него поредица от точно K числа (1<K<N) с максимална сума.
4. Напишете клас **Matrix**, който съдържа матрица от реални числа, представена чрез двумерен масив. Дефинирайте оператори за събиране, изваждане и умножение на матрици, методи за достъп до съдържанието и метод за отпечатване.
5. Напишете програма, която прочита от конзолата масив от N цели числа и цяло число K, сортира масива и чрез метода **Array.BinSearch(…)** намира най-голямото число от масива ≤ K и най-малкото число от масива ≥ K. Да се отпечата сортирания масив, с отбелязани в него търсените две числа.
6. Даден е масив от N цели числа, за който знаем, че един от елементите му (т. нар. мажорант) се среща на поне 1 + N/2 различни позиции. Да се напише програма, която с помощта на класа **Stack** намира мажоранта на масива. Например ако имаме масива {3, 2, 2, 3, 2, 1, 3, 2, 2, 2, 1}, неговият мажорант е 2. Ако се затруднявате, помислете дали не можете да обходите елементите и всеки от тях или да го добавяте в стека, ако съвпада с елемента на върха му, или в противен случай да премахвате елемента от върха на стека.
7. Даден е масив от символни низове. Да се напише метод, който намира всички низове от масива, които имат четна дължина. Методът трябва да връща масив от символни низове и трябва вътрешно да използва класа **StringCollection**.
8. Даден е масив от символни низове. Да се напише програма, която отпечатва всички различни низове от масива и за всеки от тях колко пъти се среща. Низовете в резултата трябва да са подредени по брой срещания в низходящ ред. Препоръчва се използване на хеш-таблица с ключове низовете и стойности брой срещания. За сортирането може да се използва **Array.Sort(…)**.
9. Даден е речник, който представлява масив от двойки стойности – дума и значение. Да се напише програма, която превежда поредица от думи. Има ли смисъл да се ползва хеш-таблица?
10. Да се напише клас **ComplexNumber**, който представлява комплексно число с реална и имагинерна част от тип **double**. Да се напише метод, който приема като параметър масив от комплексни числа и връща като резултат комплексното число, което се среща най-голям брой пъти в

този масив. За целта да се използва хеш-таблица, в която за всяко комплексно число се пази броят на срещанията му. Не забравяйте да реализирате предварително по подходящ начин методите **`Equals(…)`** и **`GetHashCode()`** на класа **`ComplexNumber`**.

Използвана литература

1. Светлин Наков, Масиви и колекции – http://www.nakov.com/dotnet/lectures/Lecture-7-Arrays-and-Collections-v1.0.ppt
2. MSDN Training, Programming with the Microsoft® .NET Framework (MOC 2349B), Module 7: Strings, Arrays, and Collections
3. Jeff Prosise, Programming Microsoft .NET Microsoft Press, 2002, ISBN 0735613761
4. Jeffrey Richter, Applied Microsoft .NET Framework Programming, Microsoft Press, 2002, ISBN 0735614229
5. Joel Fugazzotto, C# Complete, Sybex Inc., ISBN 0782142036
6. MSDN Library – http://msdn.microsoft.com

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 9. Символни низове (Strings)

Необходими знания

- Базови познания за общата система от типове в .NET Framework
- Базови познания за езика C#

Съдържание

- Стандартът Unicode
- Типът **`System.Char`**
- Символни низове. Класът **`System.String`**
- Escaping последователности
- Ефективно конструиране на низове чрез класа **`StringBuilder`**
- Форматиращи низове
- Класът **`StringInfo`**
- Интернационализация и култури
- Парсване на числа и дати
- Кодиращи схеми и конвертиране. Класът **`System.Text.Encoding`**
- Кодиране Base64
- Работа с Unicode във Visual Studio .NET

В тази тема ...

В тази тема ще се запознаем с начина на представяне на низовете в .NET Framework и с методите за работа с тях. Ще разгледаме какви кодиращи схеми се използват при съхраняване и пренос на текстова информация и как се решава въпросът с подредбата на байтовете. Ще се спрем подробно на различните начини за манипулиране на низове, предоставени ни от FCL, както и на някои практически съображения при работа с класовете, според решаваната задача. Ще видим как настройките за държава и култура определят вида на текста, показван на потребителите, и как можем да форматираме изхода в четлив и приемлив вид. Ще се запознаем също и с начините за преобразуване на вход от потребителя от текст в обект от стандартен тип, с който можем лесно да работим.

Стандартът Unicode

Стандартът Unicode играе много съществена роля при работата с текст в повечето съвременни софтуерни платформи. Неговата история е дълга и интересна. Той възниква в резултат от усилията за създаване на единна система за представяне на всички азбуки и езици и се налага все повече като универсално средство за представяне на текстова информация.

Преди да се запознаем със средствата на .NET Framework за работа със символни низове и текст, ще разгледаме Unicode стандарта, тъй като низовете в .NET Framework вътрешно са базирани на този стандарт.

В началото бе ASCII

В началните години от развитието си компютърната техника е съсредоточена в Северна Америка. Софтуерът по това време е бил създаван за ползване предимно в англоезични среди. Текстовите данни са били представяни чрез ASCII или EBCDIC символи. За един такъв символ е нужен 1 байт памет: за кодирането на ASCII символ - 7 бита, а за EBCDIC – 8 бита.

Постепенно в останалата част на света се разработват други системи за съхраняване на символи. В Япония това са JIS символите, за руския език се налага KOI8 стандартът, а индийските езици се представят с няколко ISCII стандарта. Изброените стандарти дефинират кодови таблици с двоично представяне на буквите, цифрите и някои други символи.

Ето как изглежда кодовата таблица, дефинирана от стандарта ASCII (American Standard Code for Information Interchange):

00-25	26-51	52-77	78-103	104-127
NUL	SUB	4	N	h
SOH	ESC	5	O	i
STX	FS	6	P	j
ETX	GS	7	Q	k
EOT	RS	8	R	l
ENQ	US	9	S	m
ACK	SP	:	T	n
BEL	!	;	U	o
BS	"	<	V	p
HT	#	=	W	q
LF	$	>	X	r
VT	%	?	Y	s
FF	&	@	Z	t
CR	'	A	[	u

SO	(	B	\	v
SI	)	C	]	w
DLE	*	D	^	x
DC1	+	E	_	y
DC2	,	F	`	z
DC3	-	G	a	{
DC4	.	H	b	|
NAK	/	I	c	}
SYN	0	J	d	~
ETB	1	K	e	DEL
CAN	2	L	f	
EM	3	M	g	

ASCII дефинира 127 символа и представя всеки от тях със 7-битово число. Тези символите включват латинските букви, цифрите, някои често използвани знаци и някои служебни символи със специално предназначение. Тъй като ASCII е бил въведен преди много време, част от специалните символи за загубили значението си.

Unicode

Unicode е проект, който има за цел да замести съществуващите символни кодови таблици. Голяма част от тях не са всеобщо приети, което създава проблеми при пренос на данни между различни среди и платформи. Въпреки техническите трудности и мащабността на проекта, Unicode се е наложил като стандарт при интернационализацията на софтуера. Той съдържа изключително богат набор от символни дефиниции. Unicode е приет и като основа за представяне на текст в много операционни системи, платформи и стандарти (XML, Java, .NET Framework и др.).

Символите в Unicode

Unicode е стандарт, предоставящ уникален номер за всеки един знак (букви, йероглифи, математически символи и др.) с цел универсалност при съхраняването им в цифров вид при различни операционни системи и езикови среди.

Версия 4.0 на Unicode стандарта дефинира близо 100 000 символа и може да поддържа над 1 милион различни знака (чрез комбинация от символи).

Стандартът Unicode се развива постоянно под контрола на Unicode консорциума (www.unicode.org) – добавят се нови символи, утвърждават се нови спецификации и т. н.

Обикновено Unicode символите се записват с "U+" и съответния номер в шестнайсетичен вид.

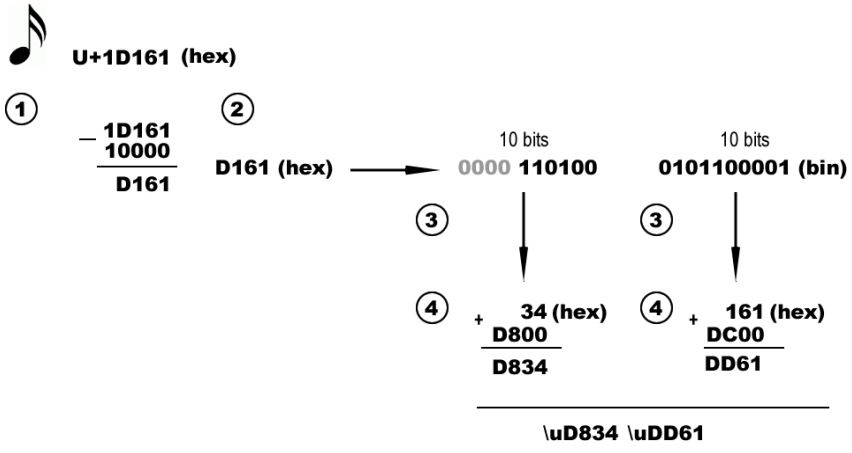
Например символът "листо" на традиционен китайски се записва като **U+8449**. Това съответства на десетичния номер 33865. За да представим този символ в сорс код на C# трябва да използваме шестнайсетичния номер и да укажем, че това е Unicode символ, чрез записа "**\u8449**". Същият символ в езика HTML се записва като **葉** или **葉**.

Да разгледаме още един пример за Unicode символ – музикалният знак "нота шестнайсетина". Стандартно този символ се записва в Unicode като **U+1D161**, което съответства на десетичния номер 119137. В HTML можем отново да използваме два записа, съответстващи на шестнайсетичния и десетичния номер **𝅘𝅥𝅯** и **𝅘𝅥𝅯**. При записа на този символ в сорс код на C# се използва т. нар. кодова двойка и символът се изписва като последователност от две шестнайсетични числа – "**\uD834\uDD61**".

Употребата на кодови двойки е необходима при символи, чиито Unicode номера не могат да бъдат кодирани само с 16 бита. Обърнете внимание, че Unicode поддържа над 500 000 символа, а в 16 бита могат да се запишат само 65536 различни стойности.

Кодови двойки

За да получим кодовата двойка, съответстваща на даден символ, трябва от шестнайсетичния му Unicode номер да извадим 0x10000, след което да преобразуваме получената стойност в двоично число и да я разделим на две части – водещите 6 бита и останалите 10 бита. Първите 6 бита, отново превръщаме в шестнайсетично число и към тях прибавяме специалния Unicode номер **U+D800**. Подобна е процедурата и с вторите 10 бита, но там прибавяме **U+DC00**. Извършването на тази операция върху символа нота шестнайсетина е илюстрирано на следната фигура:



Познавайки тези правила, лесно можем да определим дали дадена шестнайсетична стойност съответства на Unicode номер или е част от кодова двойка, както и да съобразим коя част от кодовата двойка представлява.

Като правило по-често използваните знаци в Unicode стандарта се представят с една шестнайсетбитова стойност, а останалите – с кодови двойки.

Графични знаци (графеми)

Графеми се наричат всички графични знаци от писмеността на различните езици, използвани в човешката цивилизация – букви, цифри, идеограми, пунктуационни знаци, математически символи и т. н.

Не всички графеми имат свой уникален номер в Unicode стандарта, защото някои графеми могат да се представят като комбинация от няколко други. Това е направено с цел да се намали общият брой на Unicode символите.

Например графемата **A** се представя с Unicode номер **U+0041**, но **Ạ** (A с точка отдолу) се представя с последователността **U+0041**, **U+0323** от Unicode номера. Това се дължи на факта, че в Unicode има някои символи, които не представляват самостоятелни знаци, а допълват други символи. Такива са например ударенията, горната подчертаваща чертичка, долната подчертаваща чертичка и др.

По-нататък, в секцията "Кодиращи схеми", ще разгледаме подробно кодиращите схеми, поддържани в .NET Framework, както и по какъв начин се представя текстовата информацията в паметта.

Типът System.Char

Типът **System.Char** в .NET Framework е стойностен тип и позволява съхранението на една 16-битова Unicode стойност. В езика C# примитивният тип **char** е еквивалентен на типа **System.Char**.

В повечето случаи един символ може да се представи с една инстанция на типа **char**, но това не винаги е така. В .NET Framework символите с Unicode номер, по-голям от 65536, се представят като последователност от две инстанции на **System.Char**.

Методи за класификация на символите

Типът **System.Char** притежава статичен метод **GetUnicodeCategory(…)**, който връща информация за съответния знак (дали е малка буква, главна буква, символ за валута, математически символ и т.н.). Останалите методи, служещи за класифицирането на символите, са дадени в таблицата:

Метод	Описание
IsLetter()	Проверка дали знакът е буква (letter). Букви са не само латинските **A..Z**, но и буквите от други азбуки, като например "**Ъ**", "**Σ**" и "**ẅ**".
IsDigit()	Връща **true** ако знакът е цифра.

IsWhiteSpace()	Връща **true** ако знакът е за празно място (интервал, табулация, символ за нов ред и др.).

Методите **IsLetter()**, **IsDigit()**, **IsSeparator()** и **IsWhiteSpace()** връщат са реализирани чрез вътрешно извикване на **GetUnicodeCategory()**.

Методи за смяна на регистъра

Конвертирането на знак към главни или малки букви става чрез извикване на статичните методи **Char.ToUpper()** и **Char.ToLower()**. Резултатът от извикването на двата метода зависи от информацията за културата. Тя може да бъде подадена като параметър, а по подразбиране се използва текущата. Ролята на културата ще бъде дискутирана малко по-нататък.

Символни низове в .NET Framework

В .NET Framework символните низове (strings) представляват последователност от Unicode знаци, записани в кодиране UTF-16. Това позволява на програмистите с лекота да използват в своите приложения едновременно различни езици.

За символните низове в .NET се използва системният тип **System.String**. Той представлява неизменима (immutable) последователност от символи (инстанции на **System.Char**).

Както всички типове в .NET Framework, така и **System.String** е наследник на **System.Object**. Типът **System.String** е референтен и неговите инстанции се съхраняват в динамичната памет. Той е обезопасен за многонишково изпълнение.

В езика C# типът **System.String** може да се изписва съкратено чрез неговия псевдоним, запазената дума **string**:

```
string s = "Няма бира!";
```

Поради причини, свързани с производителността, архитектите на .NET Framework са интегрирали символните низове тясно с CLR.

Веднъж създадени инстанциите на типа **String** не могат повече да бъдат променяни. При нужда от промяна на символни низове в хода на програмата, тази особеност може да доведе до намалена производителност. За справяне с този проблем при динамична обработка на символни низове се използва класът **System.Text.StringBuilder**, на който ще се спрем малко по-късно.

Символните низове и паметта

При работа с методи, които модифицират символните низове, ние не променяме самия символен низ, а създаваме нов, в който биват отразени промените.

Нека да разгледаме следния примерен код, който демонстрира работата със символни низове:

```
static void Main()
{
   string s = "Няма бира!";
   string s2 = s;
   s = "Докараха бира!";
   s += " ... и трезви няма.";
   Console.WriteLine("s = {0}", s);
   Console.WriteLine("s2 = {0}", s2);
}
```

В началото дефинираме променлива **s** от референтния тип **System.String**, като едновременно с това й задаваме стойност "Няма бира!":

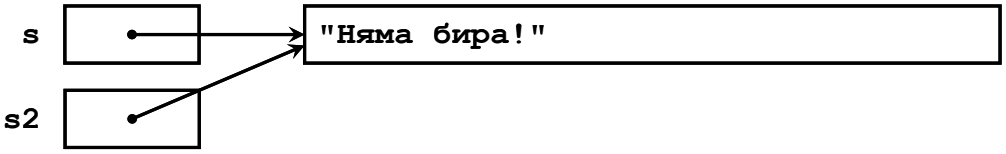
```
string s = "Няма бира!";
```

В този случай **s** е указател (референция), който сочи към определено място в динамичната памет, където е записана стойността "Няма бира!".

На следващия ред създаваме втори символен низ, който инициализираме със стойността на първия:

```
string s2 = s;
```

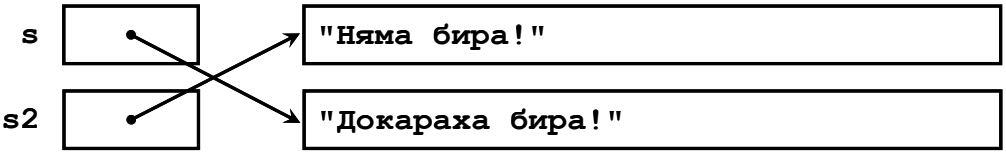
В този случай вече имаме два указателя (**s** и **s2**), които сочат към едно и също място в паметта:



При третото действие задаваме нова стойност за **s**, при което се създава нов обект в динамичната памет:

```
s = "Докараха бира!";
```

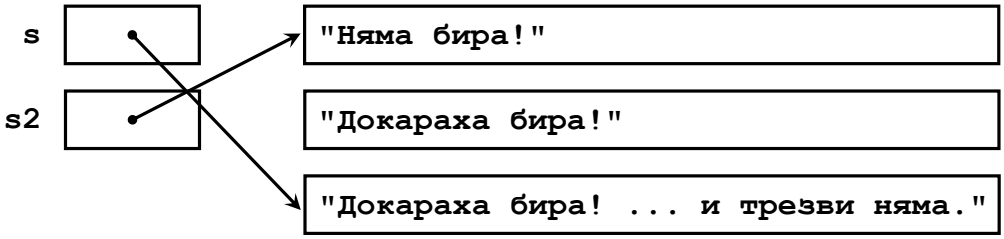
След изпълнението на този ред **s** съдържа указател, който сочи към друго място в паметта, защото стойността е променена:



Следва долепване на низа " ... и трезви няма." към стойността на низа **s**:

```
s += " ... и трезви няма.";
```

Понеже низовете в .NET Framework са неизменими, тази операция създава нов обект в динамичната памет и записва в него резултата от долепването. Старата стойност на **s** остава в динамичната памет като нереферииран от никого обект, който ще бъде освободен при следващото почистване на паметта:



Класът System.String

Класът **System.String** е основополагащ за работата със символни низове в .NET Framework. Нека се запознаем в детайли с по-важните методи и свойства на **System.String**.

Правила за сравнение на символни низове

Методите и свойствата на класа **System.String** могат да бъдат разделени на числени и лексикографски. Числените методи, изпълнявани върху символен низ, действат върху индекса (поредния номер в низа) на всеки знак. Лексикографските операции действат върху стойността на низа, съобразно избраната култура, форматиране и правила за парсване.

Много от числените и лексикографските операции се базират на правилата за сортиране. .NET Framework подържа правила за сортиране по дума, низ и последователност на Unicode символите.

Сортирането на думи (word sorting) е чувствително към културата лексикографско сравнение на низове. На Unicode символите, които не са букви или цифри, се приписва определена стойност – тегло. Например символът тире "-" може да има нулево тегло, така че низовете "coop" и "co-op" да бъдат равни. Така два низа могат да бъдат логически равни, дори и знаковите им последователности да не са еднакви. Например ако използваме немска култура, низовете "Straße" и "Strasse" ще са еквивалентни.

Сортирането на низове (string sorting) по същество е същото като сортирането по думи, с изключение на факта, че всички небуквени и нецифрени символи предхождат всички останали.

При сортирането по последователност на Unicode символите (ordinal sorting) два низа се сравняват по числената стойност (поредния номер в Unicode стандарта) на всеки символ в низа.

Процедурите за сравнение и процедурите за търсене в низове по подразбиране са чувствителни към главни и малки букви. Те използват културата, асоциирана към текущата нишка, освен ако изрично не е указана друга. По дефиниция всеки символен низ, включително и празният (""), е по-голям от стойността **`null`**. При сравнение на две стойности **`null`**, те се считат за равни.

При системи, в които се използва критично за сигурността сравнение на низове, трябва да се избере подход, при който културата не застрашава коректния резултат от сравнението. Грешка би могла да се получи, когато текущата култура на потребителя е различна от тази на приложението. Решение на проблема е употребата на инвариантна култура (на която ще се спрем подробно след малко).

Добра практика е да се използва инвариантна култура за съхранение на данни, които не се извеждат директно на потребителя. Данни за валута и дати могат да се съхраняват с инвариантна култура и при визуализирането им в приложението да бъдат форматирани по подходящ начин. Така няма да има несъответствия при изпълнението на програмата от потребители с различни настройки на културата.

Методи и свойства на System.String

По-важните свойства на класа за работа с низове са **`Length`**, което връща броя на символите в инстанцията на низа, и **`Chars[int]`**, което връща символа на зададена позиция в низа. В C# вместо свойството **`Chars[int]`** се използва за индексатор на низа **`this[int]`**, който също връща символа на зададената позиция.

Индексирането на символите в низовете в .NET Framework започва от 0. Първият символ в даден низ `s` се намира на позиция 0, а последният – на позиция `s.Length-1`.

Методи за сравнение на низове

За сравнение на низове се използва методът **`Compare(…)`**, който е предефиниран с различни по брой и/или тип параметри. При **`Compare(…)`** сравнението на низовете е лексикографско, като се взима предвид текущата културата. Като параметър може да бъде зададена специфична култура или инвариантност спрямо културата. Ако целта на сравнението е да бъдат сортирани низове, то трябва да укажем да се прави разлика между малки и главни букви. В противен случай два низа, съдържащи съответно малка и главна буква, се считат за равни и при всяко изпълнение на програмата низовете могат да бъдат сортирани в различен ред.

За разлика от метода **Compare(…)**, статичният метод **CompareOrdinal(…)** не отчита културата. При него се сравняват само Unicode стойностите и той връща равенство, ако двата низа съдържат една и съща последователност от знаци. Тъй като малките и главните букви имат различни Unicode стойности, то **CompareOrdinal(…)** е чувствителен към разлика в низовете, дължаща се на малки и главни букви.

Методите **Compare(…)** и **CompareOrdinal(…)** връщат положително, отрицателно число или 0 в зависимост дали първият аргумент е съответно по-голям, по-малък или равен на втория. Ето един пример, който илюстрира лексикографско сравнение на низове:

```
string s1 = ".NET Framework";
string s2 = ".NET Platform";
int compareResult = String.Compare(s1, s2);
Console.WriteLine(compareResult); // Prints -1
```

В примера не е указано какво култура да бъде използвана, заради което се подразбира текущата.

Друг метод за сравнение на низове е **Equals(…)**. Преди да извърши сравнението на два низа чрез вътрешно извикване на методът **CompareOrdinal(…)**, този метод проверява референциите на двата низа. Ако референциите сочат към един и същ обект в паметта, **Equals(…)** връща **true** без да прави по-нататъшна проверка. Това прави методът изключително бърз при използване на интерниране на низове, с което ще се запознаем по-късно.

Методът **Equals(…)** може да бъде извикван с различен тип и брой параметри. Той приема един или два параметъра, които могат да бъдат низове, произволен обект или инстанция на класа **StringBuilder** (който ще разгледаме след малко).

Освен методите за сравнение на низове се използват и операторите **==** и **!=**. Двата оператора са имплементирани на базата на метода **Equals(…)** на **System.String** и са чувствителни към малки и главни букви.

Търсене на съвпадения в началото и в края

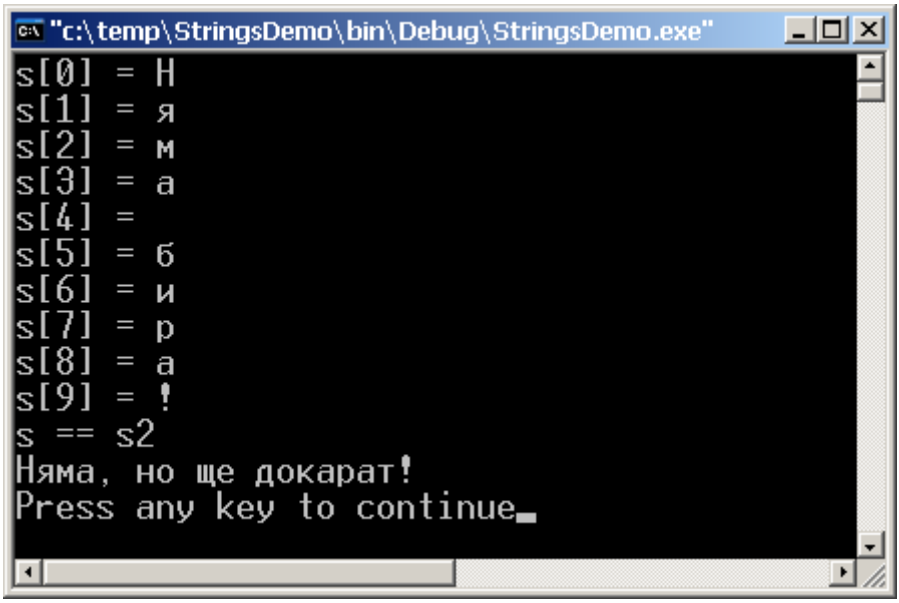
StartsWith(…) и **EndsWith(…)** са методи, които проверяват дали дадената инстанция на класа **String** започва или завършва със зададения като параметър низ. **StartsWith(…)** връща **true**, ако зададеният низ се среща в началото на инстанцията или ако сме задали празен низ, и **false** в противен случай. **EndsWith(…)** връща **true**, ако инстанцията на обекта от класа **System.String** завършва със зададения като параметър низ или ако сме задали празен низ. За целта методът сравнява параметъра с подниз, извлечен от края на текущата инстанция със същата дължина, като параметъра. И двата метода извършват търсене на дума (word search), използвайки текущата култура и са чувствителни към малки и главни букви.

Работа с низове – прост пример

Следният примерен код илюстрира индексиране на низ, слепване на низове и сравняване на низове с оператор, както и метода **`StartsWith(…)`**:

```
static void Main()
{
   string s = "Няма бира!";
   for (int i=0; i<s.Length; i++)
   {
      Console.WriteLine("s[{0}] = {1}", i, s[i]);
   }

   string s2 = "Няма " + "бира!";
   if (s == s2) // true
   {
      // Both strings have equal values
      Console.WriteLine("s == s2");
   }

   if (s.StartsWith("Няма")) // true
   {
      Console.WriteLine("Няма, но ще докарат!");
   }
}
```

Ето резултата от изпълнението на примера:



Извличане на подниз

За извличане на подниз в класа **System.String** е предвиден метод **Substring(…)**. Той може да бъде извикан с един или два параметъра от тип **Int32**, които задават съответно началото и дължината на подниза. Ако подаденият начален индекс е равен на дължината на низа и вторият параметър е 0 или липсва, то върнатият подниз ще е празен. Ако параметърът, указващ началото, е по-малък от 0 или по-голям от дължината на низа, се получава изключение **ArgumentOutOfRangeException**. Това изключение ще получим и ако подадената дължина е по-голяма от дължината на низа. Отново ще напомним, че индексирането на символите в низа започва от 0.

Търсене в низ

Методът **IndexOf(…)** връща позицията на първото срещане на посочения като параметър низ в текущия низ. Ако низът не бъде открит, върнатата стойност е **-1**. Този метод също е предефиниран по сигнатура и може да приеме като параметър **String** или **Char** обект за търсене. Може да бъде зададена и специфична област от низа, в която да се търси зададения подниз или символ. Сравнението при търсене на подниза е чувствително към малки и главни букви, но е нечувствително към културата.

Аналогично методът **LastIndexOf(…)** връща позицията на последното срещане на зададения низ в низовия обект или **-1** ако няма съвпадение.

Търсене в низове – пример

За да илюстрираме търсенето в низове, ще дадем за пример следния код, показващ действието на методите **Substring(…)** и **IndexOf(…)**:

```
string s = "Няма бира!";

string beer = s.Substring(5, 4); // beer = "бира"
Console.WriteLine(
   "\"{0}\".Substring(5,4) is \"{1}\".", s, beer);
// Result: "Няма бира!".Substring(5,4) is "бира".

int beerIndex = s.IndexOf("бира"); // 5
Console.WriteLine(
   "The index of \"{0}\" in \"{1}\" is {2}.",
   "бира", s, beerIndex);
// Result: The index of "бира" in "Няма бира!" is 5.

int vodkaIndex = s.IndexOf("водка"); // -1
Console.WriteLine(
   "The index of \"{0}\" in \"{1}\" is {2}.",
   "водка", s, vodkaIndex);
// Result: The index of "водка" in "Няма бира!" is -1.
```

Разделяне и слепване на низове

За разделяне и слепване на низове в класа **`System.String`** са предвидени няколко метода. Методът **`Split(…)`** разделя низовия обект на множество от поднизове по зададен като параметър масив от разделителни символи. Поднизовете се връщат в масив от низове (**`string[]`**).

Методът **`Join(…)`** на класа **`String`** служи за слепване на низове. При зададен разделител и масив от низове, той създава един обединен низ, като между елементите от масива постави разделителя.

Допълнителни ресурси при визуализация на низове

При визуализиране на низове в .NET често се използва методът **`Format(…)`**. Той замества всеки форматиращ идентификатор в низа със стойностите на посочените обекти. Има възможност за поставяне на няколко форматиращи идентификатора, както и задаване на специфична култура за визуализацията на стойностите. На форматиращите низове ще се спрем в детайли след малко.

За подравняване на низове при визуализация се използват методите **`PadLeft(…)`** и **`PadRight(…)`**. **`PadLeft(…)`** подравнява низа като отляво оставя посочения брой празни места или друг зададен Unicode символ. Аналогично **`PadRight(…)`** визуализира низа като оставя отдясно зададен брой празни места или зададения символ.

Разделяне, слепване и визуализация на низове - пример

След като се запознахме с методите **`Split(…)`**, **`Join(…)`** и **`Format(…)`** ще използваме следния пример, за да демонстрираме работата с тях:

```
string wisdom = "Петър плет плете, през три пръта " +
                "преплита. Подпри, Петре, плета!";
string[] words = wisdom.Split(' ', ',', '.', '!');
foreach (string w in words)
   {
      if (w != "")
      {
         Console.Write("{0} ", w);
      }
   }
Console.WriteLine();
// Result: Петър плет плете през три пръта преплита Подпри Петре
плета

string[] drinks = {"бира", "ракия", "вино", "водка"};
string wordsList = String.Join(", ", drinks);
Console.WriteLine("Остана само: {0}.", wordsList);
// Result: Остана само: бира, ракия, вино, водка.

string nonsense = String.Format(
```

```
          "Доказано е, че {0} + {1} = {2}, но само при " +
          "достатъчно големи стойности на {3}.", 2, 2, 5, 2);
Console.WriteLine(nonsense);
// Result: Доказано е, че 2 + 2 = 5, но само при достатъчно
големи стойности на 2.

string formattedDate = String.Format(
          "Днес е {0:d.MM.yyyy} г.", DateTime.Now);
Console.WriteLine(formattedDate);
// Result: Днес е 2.08.2004 г.

const int SIZE = 15;
string formattedName = "Бай Иван".PadLeft(SIZE, ' ');
string formattedSum  = "131.7 лв.".PadLeft(SIZE, '0');
Console.WriteLine("Вашето име : {0}", formattedName);
Console.WriteLine("Вие дължите: {0}", formattedSum);
// Result:
// Вашето име :         Бай Иван
// Вие дължите: 000000131.7 лв.
```

Понеже в примера не е указана култура при форматирането на датата, се използва културата по подразбиране, в случая българската култура.

Модифициране на низове

Класът **System.String** притежава и някои методи за модификация на низове. Както споменахме, стойността на обект от тип **string** е непроменяема и затова всички тези методи водят до създаване на нов обект, в който връщат модифицираният низ. Използването на тези методи е подходящо при началната инициализация на низов обект, но не е препоръчително да се използват многократно в цикъл, защото това ще намали значително производителността.

Методът **Trim(…)** премахва зададените символи от началото и края на низа. Извикан без параметър методът премахва празните места в началото и края на низа. Предефинираният метод може да бъде извикан с група от символи, които да бъдат премахнати.

Методите за смяна на главни с малки букви и обратно са **ToLower()** и **ToUpper()**. За целта те използват текущата култура, ако не бъде зададена изрично друга.

За вмъкване на низ в низовия обект се използва методът **Insert(…)**. Като параметри му се задават позицията в низовия обект, където да бъде вмъкнат подниз, и подниза, който да бъде вмъкнат. Грешки възникват при изпълнение на C# код ако е зададен **null** като подниз за вмъкване или позицията за вмъкване е извън границите на низовия обект.

Класът **System.String** предлага и метод за премахване на зададен брой символи от указана позиция в низовия обект. Това става с метода **Remove(…)**. При работа с този метод трябва да внимаваме параметрите да

не излизат извън дължината на низовия обект, защото това води до възникване на изключение.

Методът **Replace(…)** служи за заместване на зададен низ със зададен друг низ. Това заместване се извършва при всяко срещане на подадения като параметър низ в низовия обект.

Модифициране на низове – пример

Ще илюстрираме работата с методите **Trim(…)**, **ToUpper()**, **Insert(…)**, **Remove(…)** и **Replace(…)** чрез следния пример:

```
string example = ", мента, мастика, коняк.";
string trimmed = example.Trim(',', '.', ' ');
Console.WriteLine(trimmed);
// Result: мента, мастика, коняк

String upDemo = trimmed.ToUpper();
Console.WriteLine(upDemo);
// Result: МЕНТА, МАСТИКА, КОНЯК

string insertDemo = upDemo.Insert(7, "боза, ");
Console.WriteLine(insertDemo);
// Result: МЕНТА, боза, МАСТИКА, КОНЯК

string removeDemo = insertDemo.Remove(11, 9);
Console.WriteLine(removeDemo);
// Result: МЕНТА, боза, КОНЯК

string replaceDemo = removeDemo.Replace(", ", " + ");
Console.WriteLine(replaceDemo);
// Result: МЕНТА + боза + КОНЯК
```

Методи на System.String

Ще обобщим накратко действието на по-важните методи на класа **String**:

Метод	Описание
Compare(…)	Сравнява два низа лексикографски, като се съобразява с особеностите на културата.
CompareOrdinal(…)	Сравнява два низа като взима числената стойност на всеки символ от Unicode таблицата.
CompareTo(…)	Сравнява низовия обект със зададен като параметър друг низ.
Concat(…)	Слепва два или повече низа или стойности на низови обекти.
Copy(…)	Създава нова инстанция на обект от тип **string** със същата стойност.

CopyTo(…)	Копира зададен брой символи от определена област в низа и ги записва в масив.
EndsWith(…)	Определя дали низът завършва със зададения низ.
Equals(…)	Определя дали два низови обекта имат една и съща стойност.
Format(…)	Замества всеки форматиращ идентификатор със стойността на зададен обект.
GetEnumerator()	Връща обект, който може да обхожда отделните знаци на низовия обект.
IndexOf(…)	Връща позицията на първото срещане на зададен низ в низовия обект.
IndexOfAny(…)	Връща позицията на първото срещане на някои от символите, зададени в масив като параметър.
Insert(…)	Вмъква зададен низ на зададена позиция.
Intern()	Ако вече не е добавен, добавя низ в "Intern pool" таблицата и връща системния указател към него. Ще разгледаме този метод в секцията "Интерниране на низове".
IsInterned()	Връща указател към зададен низ. Ако низът не е в "Intern pool" връща **null**.
Join(…)	Слепва елементите на зададен масив от низове като между всеки два елемента поставя зададен разделител.
LastIndexOf(…)	Връща позицията на последното срещане на зададен низ в низовия обект.
LastIndexOfAny(…)	Връща позицията на последното срещане на някои от символите зададени в масив като параметър.
PadLeft(…)	Подравнява низа като добавя отляво зададен брой празни места или символи.
PadRight(…)	Подравнява низа като добавя отдясно зададен брой празни места или символи.
Remove(…)	Изтрива зададен брой символи от низа, започвайки от зададена позиция.
Replace(…)	Замества всички срещания на зададен низ със стойността на зададен друг низ в нов обект от тип **string**.
Split(…)	Разделя низа на поднизове по зададен разделител и записва получените поднизове в низов масив.

StartsWith(…)	Определя дали низът започва със зададен низ.
Substring(…)	Връща подниз от низовия обект.
ToCharArray()	Връща съдържанието на низа във вид на масив от символи (**char[]**).
ToLower()	Сменя регистъра на низа към малки букви.
ToUpper()	Сменя регистъра на низа към малки букви.
Trim(…)	Премахва зададени символи от началото и края на низа.
TrimEnd(…)	Премахва зададени символи от края на низа.
TrimStart(…)	Премахва зададени символи от началото на низа.

Escaping последователности

Когато искаме да съхраним в сорс кода на програмата специален знак или низ, който не може да се изпише директно в кода (например символа за празен ред), трябва да използваме т. нар. escaping последователности. Те представляват последователност от символи със специално значение и при компилация се заместват със съответния специален символ.

В таблицата са дадени някои често използвани escaping последователности:

Изписване	Значение	Код
\n	символ за нов ред LF (line feed)	**\x0A**
\r	символ CR (carriage return)	**\x0D**
\"	символ двойни кавички	**\x22**
\t	символ табулация	**\x09**
\'	символ апостроф	**\x27**
****	символ обратна наклонена черта	**\x5C**
\0	символ null	**\x00**

В езика C# чрез escaping последователности може да се използва произволен ASCII и Unicode символ. Това става чрез следните escaping последователности:

- **\xXX** – обозначава символ с ASCII код **XX** (шестнайсетично), например **\x0A** е символ за нов ред (LF), а **\0x41** е буквата **A** (главна, латинска)
- **\uXXXX** – обозначава Unicode символ с номер **XXXX** (шестнайсетично), напр. **\u03A3** е символа **Σ**, а **\u20AC** е символа **€**

В C# символът @ пред даден низ указва, че низът е цитиран (verbatim string) и escaping последователностите в него се игнорират.

Това е удобно, когато задаваме пълния път за достъп до файл, например:

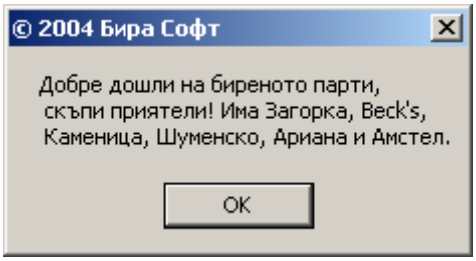
```
string fileName = @"C:\Windows\System32\calc.exe";
```

В горния пример обратната наклонена чертичка "\" не задава escaping последователности, защото низът е цитиран и се възприема без изменения, точно както е зададен.

Следва още един пример, който показва как да ползваме escaping последователности и цитиране на низове в C#:

```
static void Main()
{
   string message = @"Добре дошли на биреното парти,
                    скъпи приятели! Има Загорка, Beck's,
                    Каменица, Шуменско, Ариана и Амстел.";
   string copyright = "\xA9 2004 Бира Софт";
   System.Windows.Forms.MessageBox.Show(message, copyright);
}
```

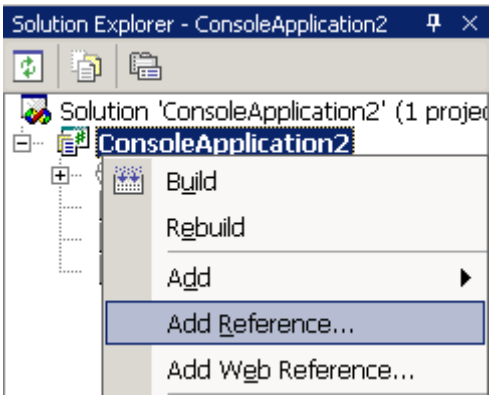
Резултатът от изпълнението на програмата е следният диалогов прозорец:

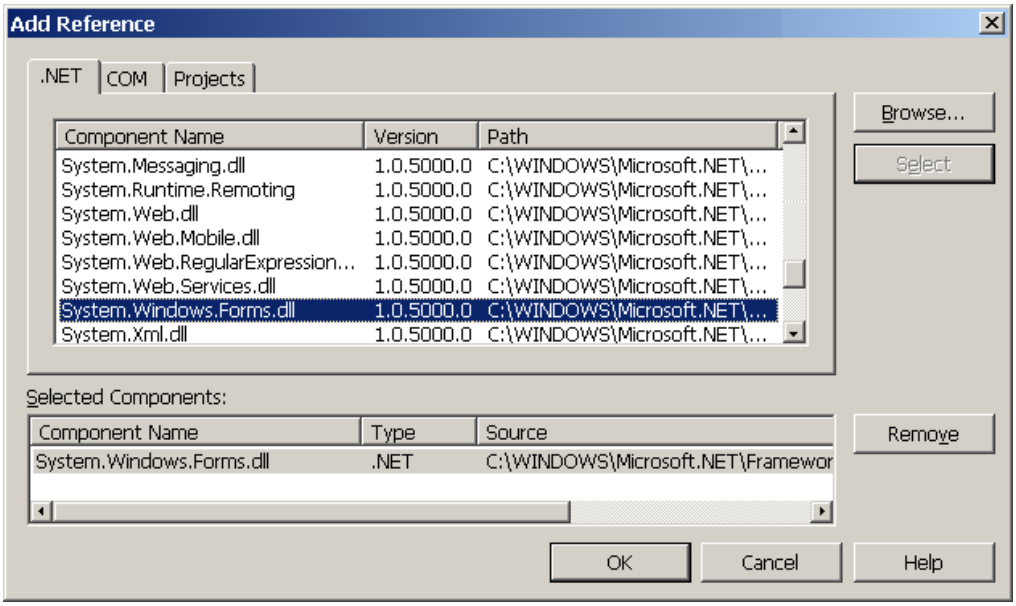


Специалният символ "©" се получава чрез escaping последователността **\xA9**. Съдържанието на диалога е цитиран низ и това запазва новите редове в текста и позволява той да бъде разположен на няколко реда в сорс кода.

В примера използвахме класа **System.Windows.Forms.MessageBox**. Този клас е дефиниран в системното асемблиу **System.Windows.Forms.dll**, което не се включва по подразбиране към проекта при създаване на конзолно приложение от VS.NET. По тази причина горният примерен код може да не се компилира успешно, ако се постави в конзолно приложение.

Ако компилаторът съобщи за грешка с текст "*The type or namespace name 'Windows' does not exist in the class or namespace 'System' (are you missing an assembly reference?)*", трябва да добавим асемблито **System.Windows.Forms.dll** към проекта. Това става като изберем "Add Reference" от контекстното меню на Solution Explorer във VS.NET и след това изберем съответното асемблиу от диалога за добавяне на референция:





Ефективно конструиране на низове чрез класа StringBuilder

Ако ви се наложи да добавяте символи към вече дефиниран низ, може би този код ще ви се стори логичен:

```
string result = "";
for (int i=0; i<10000; i++)
{
   result += "някакъв текст";
}
```

Проблемът с долепването на низове

Този код има един много сериозен недостатък. Тъй като в .NET Framework стойността на обект от тип **`string`** не може да бъде променяна след като

той веднъж е бил инициализиран, долепването на низ към друг низ е много бавна операция. Тя работи на няколко стъпки:

1. Създава се нов междинен обект (буфер) за резултата.
2. Копира се първият низ в началото на буфера.
3. Копира се вторият обект в буфера, след края на първия.
4. Създава се нов обект, символен низ и в него се копира стойността от буфера.

Старият низ и буферът остават неизползвани и се отбелязват за почистване от системата за почистване на паметта (garbage collector).

Когато изпълним процеса на долепване на низове в цикъл 10 000 пъти, се получава чувствително забавяне на програмата, което може да се избегне, ако низът се конструира по правилния начин.

Никога не конструирайте низове чрез долепване на техни части в цикъл!

Поради неизменяемостта на символните низове в .NET Framework и нуждата от допълнителен буфер, използването на операторите **+** и **+=** и методите **`Insert(…)`** и **`Remove(…)`** е изключително неефективно при многократно изпълнение. Когато е необходимо един низ да бъде променян много пъти, много по-удачно е да се ползва класът **`StringBuilder`**.

Решението на проблема – класът StringBuilder

За разлика от **`String`**, класът **`StringBuilder`** представлява изменяема последователност от символи. Той съдържа указател към масив от символи, всеки от които може да бъде променян. Могат да бъдат добавяни и нови символи в края на низа. При добавянето на нови символи се извършва проверка дали общата дължина на низа не надхвърля заделената памет (т. нар. **капацитет**). Ако дължината е по-голяма от капацитета, **`StringBuilder`** заделя нов масив, който е двойно по-голям от предишния, т. е. капацитетът се удвоява.

Горният пример може да се преработи да използва **`StringBuilder`** по следния начин:

```
StringBuilder resultBuilder = new StringBuilder();
for (int i=0; i<10000; i++)
{
   resultBuilder.Append("някакъв текст");
}
string result = resultBuilder.ToString();
```

По този начин ефективността е много пъти по-голяма, защото при всяко долепване на низ не се създават нови обекти и не се копират низове от едно място от паметта в друго. Понеже капацитетът нараства двойно при

нужда, броят изпълнения на бавната операция "преоразмеряване" е малък.

Членове на StringBuilder

В таблицата по-долу е дадено описание на по-важните свойства на класа **StringBuilder**:

Свойство	Описание
Capacity	Извлича или задава капацитета (максималният брой символи, за които има заделена памет в текущата инстанция). Обикновено **StringBuilder** заделя повече памет, отколкото му е необходима, за да не се налага преоразмеряване при добавяне или вмъкване на символи или низове. Операцията преоразмеряване е бавна, защото е свързана с преместването на всички символи на нова позиция в паметта.
Chars[int]	Взима или задава символа на зададената позиция в инстанцията. В C# това свойство е реализирано като индексатор на класа **StringBuilder**.
Length	Взима или задава дължината на инстанцията, т. е. броя символи. Чрез задаване на по-малка дължина от текущата може по ефективен начин да се премахват символи от края на текущия низ.
MaxCapacity	Връща максималния брой символи, които могат да бъдат съхранени в инстанцията (в .NET Framework 1.1 това ограничение е 2 147 483 647, т. е. 2 GB).

Методи за работа със **StringBuilder**:

Метод	Описание
Append(obj)	Добавя низ или низовото представяне на даден обект към края на инстанцията.
AppendFormat(…)	Добавя низовото представяне на обект или група обекти, форматирани по даден форматиращ низ.
EnsureCapacity(int capacity)	Обезпечава минималния брой символи, които могат да бъдат съхранени в инстанцията, като заделя памет за посочения брой символи, ако текущото количество заделена памет е по-малко от указаното.
Equals(obj)	Проверява дали даден обект е равен на текущата инстанция. Извършва се сравнение символ по символ.

Insert(index, obj)	Вмъква низово представяне на указания обект с начало указаната позиция.
Remove(int index, int lenght)	Премахва указана област от символи.
Replace(…)	Замества всеки срещнат указан символ или низ с друг указан символ или низ.
ToString(…)	Конвертира **StringBuilder** към **String**.

Използване на StringBuilder – пример

За да илюстрираме по-добре работата със **StringBuilder**, ще дадем пример за неговото използване за модифициране на символни низове. Функцията **UpcaseTextInBrackets(…)** приема като параметър низ и връща като резултат същия низ, като частта от низа, която е била в скоби, се преобразува в главни букви:

```
public static string UpcaseTextInBrackets(string aText)
{
   StringBuilder result = new StringBuilder(aText);
   int brackets = 0;
   for (int i=0; i<result.Length; i++)
   {
      if (result[i] == '(')
      {
         brackets++;
      }
      else if (result[i] == ')')
      {
         brackets--;
      }
      else if (brackets > 0)
      {
         result[i] = Char.ToUpper(result[i]);
      }
   }
   string resultStr = result.ToString();
   return resultStr;
}
```

От указания в конструктора низ се създава обект **StringBuilder**, който позволява да бъдат променяни отделните символи чрез индексатора **this**. След като приключи промяната на низа се използва **StringBuilder.ToString()**, за да се извлече променения низ във вид на **String**.

При създаването на обекта **result** от тип **StringBuilder** е използван конструкторът **StringBuilder(string)**. Чрез него на инстанцията на **StringBuilder** се задава първоначалната стойност, а като размер на

масива от символи се задава най-близката по-голяма от дължината на низа степен на 2, или 16, ако низът е по-къс от 16 знака. Например ако извикаме разгледаната в примера функция **UpcaseTextInBrackets(…)** с параметър "**Test**", размерът на заделената памет, който можем да проверим чрез свойството **Capacity**, ще е 16. Ако подадем като параметър низ с дължина от 100 символа, за масива ще бъдат заделени 128 позиции.

Заделянето на повече от указания брой позиции позволява при добавянето на неголям брой символи да се избегне заделянето на нов масив и копирането на данните в него, защото това е бавна операция. Заделяната допълнителна памет е с размер, близък до дължината на низа, за да не се използва прекомерно количество памет.

Задаване на първоначален размер за StringBuilder

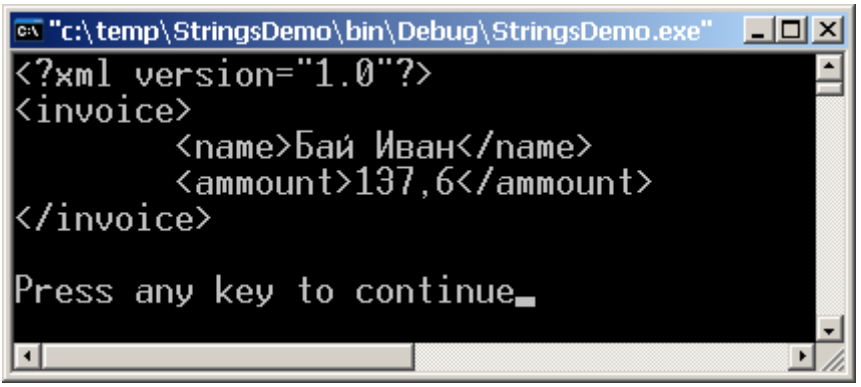
Винаги, когато се работи със **StringBuilder** и е приблизително известен очакваният размер на резултатния низ, който ще бъде конструиран, се препоръчва да се задели предварително количеството памет, необходимо за него или дори малко повече от него (да има малък резерв). Това увеличава значително производителността, защото след първоначалното заделяне на паметта не се извършва нито едно преоразмеряване по време на работа, което е много бавна операция. Началният капацитет на **StringBuilder** може да се подаде в конструктора му.

Следният пример илюстрира динамично конструиране на низ (фактура в XML формат) като използва конструктора **StringBuilder(int)**, чрез който задава първоначален размер на работния масив от символи. Така се избягва заделянето на нова памет при добавянето на текст чрез методите **Append(…)** и **AppendFormat(…)**.

```
using System;
using System.Text;

class StringsDemo
{
   static string CreateXmlInvoice(string aName, double aAmmount)
   {
      StringBuilder invoiceXml = new StringBuilder(1024);
      invoiceXml.Append("<?xml version=\"1.0\"?>\n");
      invoiceXml.Append("<invoice>\n");
      invoiceXml.AppendFormat("\t<name>{0}</name>\n", aName);
      invoiceXml.AppendFormat("\t<ammount>{0}</ammount>\n",
         aAmmount);
      invoiceXml.Append("</invoice>\n");
      return invoiceXml.ToString();
   }

   static void Main()
   {
```

```
      string invoice = CreateXmlInvoice("Бай Иван", 137.60);
      Console.WriteLine(invoice);
   }
}
```

Резултатът е изпълнението на примера е следният:



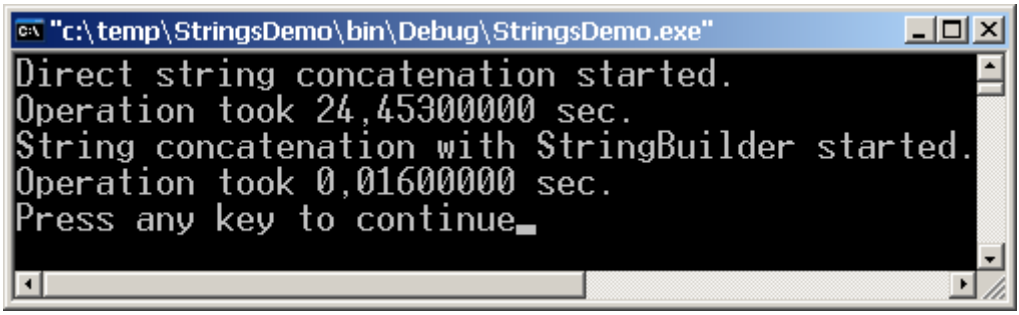
Сравнение на скоростта на String и StringBuilder – пример

За да илюстрираме разликата в производителността между долепването на низове с оператора "+" на класа **String** и долепването на низове с помощта на **StringBuilder**, ще използваме следния пример:

```
static void Main()
{
   // Append strings directly - runs very slowly
   long startTime = Environment.TickCount;
   Console.WriteLine("Direct string concatenation started.");
   string result = "";
   for (int i=0; i < 15000; i++)
   {
      result += "Дайте бира! ";
   }
   long endTime = Environment.TickCount;
   long duration = endTime-startTime;
   Console.WriteLine("Operation took {0:F8} sec.",
      (decimal)duration/1000);

   // Append strings with StringBuilder - the correct way
   startTime = Environment.TickCount;
   Console.WriteLine("String concatenation with StringBuilder
started.");
   StringBuilder resultBuilder = new StringBuilder();
   for (int i=0; i < 15000; i++)
   {
```

```
      resultBuilder.Append("Дайте бира! ");
   }
   result = resultBuilder.ToString();
   endTime = Environment.TickCount;
   duration = endTime-startTime;
   Console.WriteLine("Operation took {0:F8} sec.",
      (decimal)duration/1000);
}
```

В примера се измерва времето за изпълнение на 15000 слепвания на низ, реализиран по два начина – чрез оператора **+=** от класа **String** и по правилния начин – чрез класа **StringBuilder**. Полученият резултат е подобен на следния:



Виждаме, че има огромна разлика във времето за изпълнение на двете реализации на слепванията на 15000 низа. При първата реализация във всяко изпълнение на цикъла се създава нов обект, друг обект се маркира за унищожаване и се копират данни между двата. Тъй като тази операция се изпълнява хиляди пъти, това довежда до голямо забавяне в изпълнението. Втората реализация работи няколко хиляди пъти по-бързо, защото не изпълнява постоянно тежката операция "преместване на низ".

Понякога при изпълнението на горния пример е възможно да бъде отчетено време 0,00000000 секунди за изпълнението на втората реализация, но това се дължи на слабата прецизност на таймера, с който се отчита времето.

Класът StringInfo

Както споменахме при разглеждането на стандарта Unicode, някои символи се представят с повече от една 16-битова стойност. Тъй като един **Char** обект съдържа информация за една 16-битова стойност, няма как той да съхрани графеми, съставени от повече от една Unicode стойност.

Ако един низов обект, съдържащ съставни графеми, се обхожда символ по символ, то една графема, която се състои от няколко, примерно три, абстрактни Unicode стойности, няма да бъде коректно разпозната. Тъй като тези стойности са записани в последователни **Char** обекти, индекса-

торът на класа **String** ще разгледа всяка съставна стойност като отделен символ.

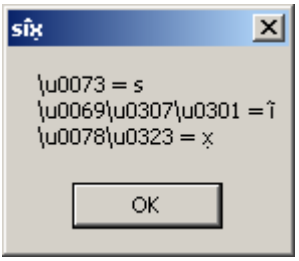
Съставните символи могат да бъдат съхранявани в низ, като класът **StringInfo** дава възможност низовете да се обхождат графема по графема. Чрез метода **GetTextElementEnumerator(…)** можем да получим итератор по отделните графеми на низа. След това с методите **MoveNext()** и **GetTextElement()** можем да осъществяваме достъп до поредната графема.

Използване на StringInfo – пример

Следващият пример илюстрира обхождането на отделните графеми от даден Unicode текст:

```
String s = "s\u0069\u0307\u0301\u0078\u0323";
TextElementEnumerator graphemes =
   StringInfo.GetTextElementEnumerator(s);
StringBuilder text = new StringBuilder();
while (graphemes.MoveNext())
{
   string grapheme = graphemes.GetTextElement();
   foreach (char ch in grapheme)
   {
      text.AppendFormat("\\u{0:X4}", (int) ch);
   }
   text.Append(" = " + grapheme);
   text.Append(Environment.NewLine);
}
MessageBox.Show(text.ToString(), s);
```

Резултатът от изпълнението на примера е низ от три символа в заглавието на показаната диалогова кутия и всяка от получените графеми, представена със съответните Unicode стойности, от които се състои.



Друг начин за ползване на класа **StringInfo** е чрез неговия метод **ParseCombiningCharacters(…)**. Той приема като параметър **String** обект и връща масив от цели числа, които съдържат началните индекси на отделните единични или съставни символи.

Интерниране на низове

При работа със **string** обекти често се налага да се извърши сравняване на два низа. Тази операция отнема много системни ресурси и ако нашето приложение предполага много такива операции, то това ще влоши бързодействието на програмата ни.

С цел спестяване на памет CLR поддържа хеш таблица на всички използвани низови константи чрез структурата от данни "**intern pool**". Всеки символен низ, които се намира в "intern pool", се нарича **интерниран**.

В тази таблица се записват всички заложени в сорс кода символни низове, като е предвиден и методът **System.Intern(String)** за добавяне на динамично генерираните символни низове.

При добавянето на нов запис в таблицата първо се проверява дали символният низ не съвпада с вече въведен такъв. Така еднаквите низове се съхраняват в паметта физически на едно място. Сравнението на два интернирани низа става много по-бързо, защото те не се сравняват по съставящите ги символи, а се извършва директно сравнение на референциите им.

Методът **System.Intern(…)** позволява на програмиста сам да интернира динамично генериран низ по време на изпълнение на програмата, ако той не съществува в таблицата, и връща референция към него.

Методът **String.IsInterned(…)** проверява дали даден низ съществува в хеш таблицата и връща референция към него, ако вече е добавен. В противен случай връща **null**.

Всички символни низове, заложени като константи в сорс кода, винаги се добавят в хеш таблицата "intern pool", но това не важи за низовете, дефинирани по време на изпълнение. Нека разгледаме следния пример:

```
string s1 = "Няма бира!";
string s2 = "Няма ";
s2 = s2 + "бира!";
Console.WriteLine(Object.ReferenceEquals(s1,s2)); // False
s2 = String.Intern(s2);
Console.WriteLine(Object.ReferenceEquals(s1, s2)); // True
```

На първия ред дефинираме **String** обект, като стойността му ще се запише в хеш таблицата на "intern pool", когато то се компилира от JIT компилатора.

Тъй като **s2** е дефиниран динамично, стойността му не се записва в "intern pool" и методът **Object.ReferenceEquals(s1, s2)** ще върне **false**.

За да интернираме **s2**, използваме статичния метод **String.Intern()**, който приема като параметър низ и търси в хеш таблицата "intern pool". Ако подаденият низ бъде намерен, методът връща референция към вече съществуващото поле.

Форматиращи низове

Възможностите на .NET Framework за форматиране ни позволяват да преобразуваме обект в низ, който е подходящ за извеждане на екрана. С форматиращи низове лесно може да представим информацията, предназначена за потребителя, в четим и разбираем вид.

Следните методи използват форматиращи низове:

- **IFormattable.ToString(string format)**
- **String.Format(string format, …)**
- **Console.WriteLine(…)**
- **StringBuffer.AppendFormat(…)**

Използване на форматиращи символи

Форматирането се задава с форматиращи символи и комбинации от тях. Ето няколко примера:

```
string s = 42.ToString("C5"); // s = "00042"
string s = String.Format("{0:P2}", 0.374); // s = "37,4%"
Console.WriteLine("Приходите за {0:d.MM.yyyy г.} са " +
  "{1:C2}", DateTime.Now, 1872.43);
// Result: Приходите за 4.08.2004 г. са 1 872,43 лв
```

Методите, които приемат съставен форматиращ низ (composite formatting string) и променлив брой аргументи, изискват параметрите да се номерират и поставят във фигурни скоби, съдържащи незадължителен форматиращ низ. Ето пример:

```
Console.WriteLine("Здравейте, {0}!\n" +
  "Днес е {1:dd.MM.yyyy, а часът е HH:mm:ss}.\n" +
  "Вие сте посетител номер {2:D6}.\nДължите {3:C2}.",
  "Бай Иван", DateTime.Now, 38, 17.40);
// Result:
// Здравейте, Бай Иван!
// Днес е 04.08.2004, а часът е 19:01:09.
// Вие сте посетител номер 000038.
// Дължите 17,40 лв.
```

В примера методът **Console.WriteLine(…)** се извиква с форматиращ низ и 4 параметъра, които се използват при форматирането.

Форматиране на числа

Стандартните форматиращи низове позволяват задаването на изгледа при отпечатване на основните типове числови стойности. Форматиращият низ трябва да е във вида **Axx**, където **A** е символ от азбуката, наречен **форматен спецификатор** (format specifier), а **xx** е незадължително число

между 0 и 99 (**указател за точност**), което задава броя на значимите цифри или броя на нулите след десетичната запетая. Форматиращият низ не трябва да съдържа празни места.

Форматният спецификатор трябва да е един от позволените форматиращи символи (по-важните от тях са изброени в таблицата по-долу). В противен случай се получава изключение от тип **`FormatException`**.

Спецификатор на форматиране	Име	Описание
`C` или **`c`**	валута	Числото се преобразува в низ, който представлява парична стойност в стандартния формат за валута на текущата нишка (ако не бъде зададен изрично друг формат за валута – **`NumberFormatInfo`** обект).
`D` или **`d`**	десетична стойност	Числото се преобразува в низ от десетични цифри (0-9), предхождани от минус, ако е отрицателно. Указателят за точност определя минималния брой цифри, присъстващи в получения низ. При необходимост се добавят нули от ляво. Този формат се поддържа само от целочислени типове.
`E` или **`e`**	научен (експоненциален)	Числото се преобразува в експоненциален формат. Например числото 3.14 се записва във вида 3,140000E+000. Указателят за точност определя броя на знаците след десетичната запетая.
`F` или **`f`**	фиксирана запетая	Подаденото реално число се записва като низ с фиксирана запетая, например "3,14". Указателят за точност определя броя на десетичните знаци. В зависимост от културата се използва различен разделител между цялата и дробната част ("." или ",").
`N` или **`n`**	число	Числото се преобразува в низ от вида "-d,ddd,ddd.ddd…", където всяко ‘d’ обозначава цифра (0-9). Низът започва със знака минус, ако числото е отрицателно. Между всяка група от три цифри в ляво от десетичната запетая се поставя разделител. Указателят за точност задава желания брой знаци след десетичната запетая.
`P` или **`p`**	процент	Числото се преобразува в проценти чрез умножение по 100. Указателят за точност задава желания брой десетични знаци.

X или **x**	шестнайсетично число	Числото се преобразува в низ от шестнайсетични цифри. Например числото 234 се записва като "EA". Това форматиране се поддържа само от целочислени типове.

Трябва да имаме предвид, че при повечето форматиращи низове крайният вид на низа зависи от настройките в Regional Options в Control Panel, когато не е подадена изрично култура. Класът **NumberFormatInfo** съдържа информация за валута, десетични разделители и други символи, свързани с числовите стойности. Ще го разгледаме по-подробно в секцията за интернационализация и култури.

Следните примери демонстрират форматирането на число във вид на валута, експоненциален запис, шестнайсетичен запис и процент.

```
int i = 100;
string formatted = i.ToString("C");
Console.WriteLine(formatted);
// Result: 100,00 лв
// (при български настройки на Windows)

double d = -3.27e38;
Console.WriteLine("{0:E}", d);
// Result: -3,270000E+038

int value = 29690;
Console.WriteLine("0x{0:X8}", value);
// Result: 0x000073FA

double p = 0.3378;
Console.WriteLine("{0:P}", p);
// Result: 33,78 %
```

В горния пример видът на получения низ зависи от регионалните настройки на машината, на която е стартиран.

Форматиране със собствени шаблони

При отпечатване на числа имаме възможност да задаваме и собствени шаблони за форматиране чрез използването на следните символи:

Форматиращ знак	Име	Описание
#	диез	Замества една цифра, като при липса на цифра не се отпечатва нищо.
0	нула	Замества една цифра, като при липса на цифра се отпечатва 0.
.	точка	Задава десетичната запетая.

,	запетая	Задава разделител между хилядите.

Следният пример показва как могат да се ползват тези символи за създаване на собствени шаблони:

```
long tel = 359888123456;
Console.WriteLine("Тел. {0:(+###) (##) ### ## ##}", tel);
// Result: Тел. (+359) (88) 812 34 56

decimal sum = 4317.60M;
Console.WriteLine("Sum = {0:###,###,##0.00}", sum);
// Result: Sum = 4 317,60
```

Както и в предходния пример, видът на получения низ зависи от регионалните настройки на машината, на която е стартиран. Те задават вида на десетичната точка, както и разделителите между хилядите.

Отместване при форматирането

При отпечатване с форматиране можем да задаваме ширината на полето, в което трябва да се запише резултатът. Ширината се указва след номера на аргумента, отделена със запетая. Например **{0,10}** разполага нулевият аргумент в пространство от 10 символа, като допълва отляво с интервали, а **{0,-8}** разполага нулевият аргумент в пространство от 8 символа, като допълва отдясно с интервали.

```
decimal sum = 4317.60M;
Console.WriteLine("Sum = |{0,16:C}|", sum);
// Result: Sum = |     4 317,60 лв|

string ip = "62.44.14.203";
Console.WriteLine("IP: |{0,-20}|", ip);
// Result: IP: |62.44.14.203        |
```

Освен чрез стандартните форматиращи низове, начинът на отпечатване на стойностите може да се контролира и чрез потребителско форматиране.

В горния пример, понеже се използва типът **decimal**, константата, която му се присвоява като стойност, трябва да завършва с буквата "**M**". В езика C# тази буква указва, че константната стойност е от тип **System.Decimal**. Ако тази буква се пропусне, компилаторът ще счита константата за число от тип **double**. По подобен начин константите от тип **long** трябва да завършват на "**L**", а константите от тип **float** – на "**F**".

Форматиране на дати и часове

Форматирането на дати и часове се извършва чрез шаблони за форматиране. Тези шаблони представляват символни низове, които се състоят от

форматиращи символи (букви от латинската азбука, които се заместват с елемент от подадената дата или час) и други символи, които се използват без изменение. В таблицата са дадени най-често използваните форматиращи символи:

Форматиращ символ	Описание
d, dd	Замества деня от подадената дата (във вариант съответно с и без водеща нула).
M, MM	Замества месеца от подадената дата (във вариант съответно с и без водеща нула).
MMMM	Замества пълното име на месеца от подадената дата на езика, свързан с културата на текущата нишка.
yy, yyyy	Замества годината от подадената дата (в двуцифрен или четирицифрен формат).
h, hh, H, HH	Замества часа от подадените дата и време (във вариант съответно с и без водеща нула). Във варианта с малки букви се използва 12-часов формат, а във варианта с главни букви - 24-часов формат.
m, mm	Замества минутата от подадените и време (във вариант съответно с и без водеща нула).
s, ss	Замества секундата от подадените и време (във вариант съответно с и без водеща нула).
/	Замества текущия разделител между дни, месеци и години. Разделителят се взима от текущата култура (от свойството **DateTimeFormatInfo.DateSeparator**).
:	Замества текущия разделител между часове, минути и секунди. Разделителят се взима от текущата култура (от свойството **DateTimeFormatInfo.TimeSeparator**).

При отпечатване на дата може да се зададе или да не се зададе шаблон. Ако шаблон не бъде зададен, се използва шаблонът по подразбиране за текущата култура. От текущата култура зависят и някои от форматиращите символи, например разделителите.

Форматиране на дати и часове – пример

Ще демонстрираме използването на форматиращи низове за дати с няколко примера:

```
DateTime now = DateTime.Now;
Thread.CurrentThread.CurrentCulture=CultureInfo.InvariantCulture;
Console.WriteLine("{0:d.MM.yyyy - HH:mm:ss} ", now);
// Result: 28.09.2005 - 19:25:02
```

```
Thread.CurrentThread.CurrentCulture = new CultureInfo("bg-BG");
DateTime birthDate = new DateTime(1980, 06, 14, 06, 10, 00);
Console.WriteLine(
   "Аз съм роден на {0:d MMMM yyyy г., в HH:mm часа}.",
   birthDate);
// Result: Аз съм роден на 14 Юни 1980 г., в 06:10 часа.

Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
Console.WriteLine(DateTime.Now);
// Result: 9/28/2005 7:24:49 PM
```

Потребителско форматиране

Типовете могат да дефинират свое собствено форматиране. За целта е нужно типът да имплементира интерфейса **`IFormattable`** и да предоставя свой метод **`ToString(string format, IFormatProvider formatProvider)`**.

Възможно е също да си дефинираме собствен метод, който да се грижи за форматиране на произволен тип. За да направим наш метод **`Format(…)`**, който да използваме за форматирането на даден тип, трябва да имплементираме **`ICustomProvider`** и **`IFormatProvider`**.

Потребителско форматиране за собствени типове

Можем да добавим възможност за обработка на дефинирани от нас форматиращи низове в дефинираните от нас собствени типове. За целта трябва в нашите типове да имплементираме интерфейса **`IFormattable`** и метода му **`ToString()`**. По този начин ще имаме контрол върху това какви форматиращи низове ще разпознава нашият тип.

Методът **`ToString(…)`** на **`IFormattable`** приема като параметри форматиращ низ и **доставчик на формат** (format provider). Ако форматиращият низ е празен или **`null`**, се извършва форматиране по подразбиране. Ако доставчикът на форматиране е **`null`**, се използва този по подразбиране.

Следва пример за тип, поддържащ няколко вида форматиращи низове:

```
class Money : IFormattable
{
   private double mAmmount;

   public Money(double aAmmount)
   {
      mAmmount = aAmmount;
   }

   public string ToString(string aFormat,
      IFormatProvider aFormatProvider)
   {
```

```
    if (aFormat == "USD")
    {
      return String.Format(
        "${0:###,###,##0.00}", mAmmount);
    }
    else if (aFormat == "BGL")
    {
      return String.Format(
        "{0:###,###,##0.00} лв.", mAmmount);
    }
    else
    {
      throw new FormatException(String.Format(
        "Invalid format: {0}", aFormat));
    }
  }

  static void Main()
  {
    Money m = new Money(27000);
    Console.WriteLine("В долари: {0:USD}", m);
    // Result: В долари: $27 000,00
    Console.WriteLine("В лева: {0:BGL}", m);
    // Result: В лева: 27 000,00 лв.
    try
    {
      Console.WriteLine("В евро: {0:EUR}", m);
    }
    catch (FormatException fe)
    {
      Console.WriteLine("Error: {0}", fe.Message);
    }
    // Result: Error: Invalid format: EUR
  }
}
```

Класът **Money** от примера реализира интерфейса **IFormattable** и може да бъде форматиран с низовете "**USD**" и "**BGL**".

Потребителско форматиране за съществуващи типове

Можем да управляваме форматирането на вече съществуващите типове, като създаваме собствени класове, доставчици на формат. Те трябва да имплементират интерфейсите **ICustomFormatter** и **IFormatProvider**. При извикването на метода **ToString(…)**, правилата, по които да ще се извърши форматирането, се определят от подадения като параметър доставчик на формат. Няма да се спираме не повече детайли. Конкретен пример за използването на **ICustomFormatter** и **IFormatProvider** има в MSDN Library, в помощната информация за класа **IFormatProvider**.

Интернационализация

Класовете **String** и **Char** зависят от т. нар. култура, която използват. Техните методи **Format(…)**, **Compare(…)**, **ToUpper()** и **ToLower()** работят по различен начин за различните езици. Културата влияе и на стандартните форматиращи низове, чрез които се форматират числа, валути и др.

Културите се дефинират от класа **System.Globalization.CultureInfo** и всяка нишка в приложението има асоциирана с нея култура (инстанция на този клас).

Културата на текущата нишка може да бъде извлечена от свойството **CurrentCulture** на класа **System.Threading.Thread**. Ако не бъде указана изрично, по подразбиране текущата култура се извлича от регионалните настройки в контролния панел на Windows.

Културите в .NET Framework

Една култура съдържа информация за календар, формат на дати и часове, формат на числа и валути, таблица за сравнение на низовете, таблица за промяна на регистъра на буквите и др.

Инстанция на класа **CultureInfo** може да получим чрез конструктора **CultureInfo**, на който се подава код на език, следван от код на държава:

```
CultureInfo cultureFrenchCanada = new CultureInfo("fr-CA");
CultureInfo cultureEngUnitedStates = new CultureInfo("en-US");
```

Друг начин за инстанциране на класа **CultureInfo** е с помощта на статичното му свойство **InvariantCulture**, което връща инвариантната (общата) култура:

```
CultureInfo cultureNeutral = CultureInfo.InvariantCulture;
```

Инвариантната култура не е свързана с никой език и държава и се използва, за работа в неутрална среда, примерно за записване на числа в неутрален формат, който не зависи от държавата.

Класът CultureInfo

Класът **CultureInfo** се използва при четене на текстови ресурси, форматиране, парсване, смяна на регистъра, сортиране и други операции, зависещи от настройките на потребителя за език и държава.

Свойства на класа CultureInfo

По-важните свойства на класа **CultureInfo** са **Calendar**, **CompareInfo**, **DateTimeFormat**, **NumberFormat** и **TextInfo**:

- **Calendar** – съдържа календара на културата (григориански, юлиански, еврейски, японски, корейски, тайвански), включващ информация за ерите, високосните години, броя месеци и броя дни за всеки месец в зависимост от годината.
- **CompareInfo** –дефинира как да бъдат сравнявани низове за зададената култура.
- **DateTimeFormat** – съдържа формата за визуализация на дата и час за зададената култура.
- **NumberFormat** – съдържа формата за запис на числата и вида на валутата.
- **TextInfo** – дефинира писмеността, асоциирана с текущата култура. Писмеността определя правилата за смяна на регистъра.

За всяко от тези свойства има съответни класове, дефинирани в пространството от имена **System.Globalization**.

Класът CultureInfo – пример

Със следния пример ще демонстрираме влиянието на културата при форматиране на валута:

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

class CulturesDemo
{
   static void ShowAmmount(
     double aAmmount, CultureInfo aCulture)
   {
     string message = String.Format(
        aCulture, "Ammount: {0:C}", aAmmount);
     String caption = aCulture.EnglishName;
     MessageBox.Show(message, caption);
   }

   static void Main()
   {
     ShowAmmount(137.25, new CultureInfo("bg-BG"));
     ShowAmmount(137.25, new CultureInfo("en-US"));
     ShowAmmount(137.25, new CultureInfo("de-DE"));
   }
}
```

В диалоговата кутия ще се визуализира сумата 137.25, като валутата ще бъде съответно в лева, долари и евро:

Списък на културите – пример

Следният код реализира визуализиране на всички вградени в .NET Framework култури:

```
using System;
using System.Globalization;

public class ListAllCulturesDemo
{
   public static void Main()
   {
      CultureInfo[] allCultures = CultureInfo.
         GetCultures(CultureTypes.AllCultures);
      foreach (CultureInfo ci in allCultures)
      {
         Console.Write("{0,-11}", ci.Name);
         Console.Write(" {0,-4}", ci.TwoLetterISOLanguageName);
         Console.Write(" {0,-4}",
            ci.ThreeLetterWindowsLanguageName);
         Console.WriteLine(" {0,-30}", ci.DisplayName);
      }
   }
}
```

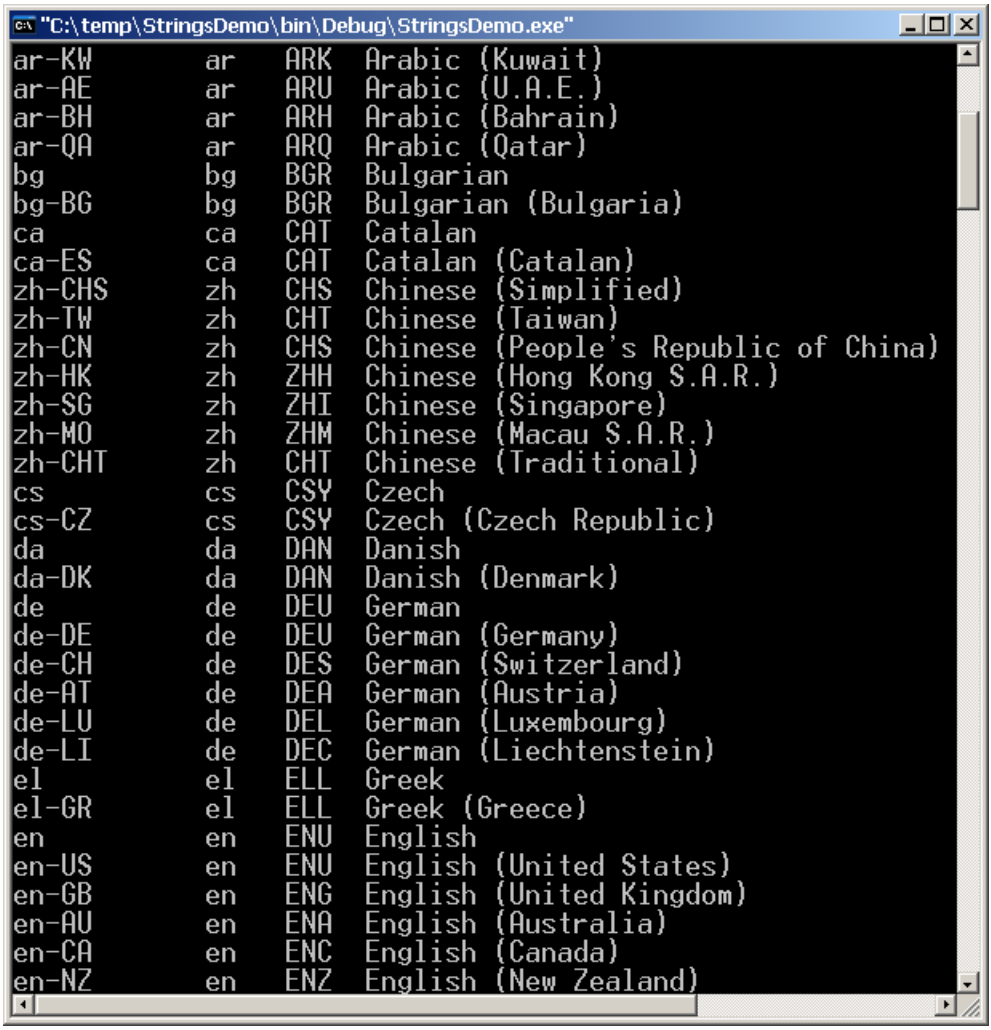
Извличане на списък от всички култури в .NET Framework – пример

Целта на следващия пример е да изведем списък от всички култури, поддържани от .NET Framework. Списъкът ще бъде съставен от няколко колонки - .NET идентификатор на култура, двубуквен ISO идентификатор, Windows идентификатор и текстово описание. Ето как изглежда сорс кода на примера:

```
using System;
using System.Globalization;

public class ListAllCulturesDemo
{
   public static void Main()
   {
```

```
      Console.WriteLine("CULTURE     ISO  WIN  DISPLAYNAME");
      Console.WriteLine("-------     ---  ---  -----------");
      CultureInfo[] allCultures =
        CultureInfo.GetCultures(CultureTypes.AllCultures);
      foreach (CultureInfo ci in allCultures)
      {
        Console.Write("{0,-11}", ci.Name);
        Console.Write(" {0,-4}", ci.TwoLetterISOLanguageName);
        Console.Write(" {0,-4}",
          ci.ThreeLetterWindowsLanguageName);
        Console.WriteLine(" {0,-30}", ci.DisplayName);
      }
    }
}
```

Ето и извадка от получения резултат:



Парсване на типове

Задачата за получаване на обект въз основа на низ е често срещана в практиката. Низът може да е получен като вход от потребителя или взет от база данни. Целта ни е да конструираме обект от стойността му, за да бъде тя по-лесно използваема в нашето приложение.

В .NET Framework за решението на тази задача е предоставено специално средство – методът **Parse(string)**, който прави синтактичен анализ (парсване). Механизмът на неговото действие е следният: на метода **Parse(string)** на обект, поддържащ парсване, се подава низ, който задава валидна стойност за инстанция на такъв обект. Методът **Parse(string)** обработва подадения низ и връща инстанция на обекта със зададената стойност. Така методът изпълнява ролята на конструктор при създаването на обекта.

Методът **Parse(string)** реализира противоположната функция на метода **ToString()**. Подобно на него, той използва обект, доставчик на формат (**FormatProvider**), за да определи какво тълкуване да направи на подадения му символен низ.

Методът **Parse(string)** се поддържа стандартно от всички числени типове, от типа **DateTime**, от изброените типове (enums), както и от типовете **Char** и **Boolean**.

Парсване на числови типове

Методът **Parse(string)** може да парсва числа, записани в различни формати. Вариантът **Parse(string, NumberStyles)** позволява да се задават различни опции при парсването. **NumberStyles** дефинира битово поле, в което всеки бит обозначава дали е позволено или не определено поведение на входящия низ. В таблицата са дадени по-важните флагове и значението, което **NumberStyles** им придава:

Име на флага	Описание
None	Специални знаци не са позволени.
AllowLeadingWhite, **AllowTrailingWhite**	Указва, че при парсването трябва да се пренебрегнат празни места в началото / в края.
AllowLeadingSign	Указва, че числовият низ може да има знак в началото. Валидните символи за знак се определят културата, от свойствата **PositiveSign** и **NegativeSign** на **NumberFormatInfo**.
AllowParentheses	Указва, че числовият низ може да бъде заграден в двойка скоби.
AllowDecimalPoint	Указва, че числовият низ може да съдържа десетична запетая. Валидните знаци за десе-

	тична запетая се определят от свойството **NumberFormatInfo** на използваната култура.
AllowThousands	Указва, че числовият низ може да съдържа групиращи разделители - разделящи например стотиците от хилядите. Валидните разделители се определят от **NumberFormatInfo** свойството на използваната култура.
AllowExponent	Указва дали числовият низ може да е в експоненциален запис.
AllowCurrencySymbol	Указва, че числовият низ се приема като валутна информация, ако има наличен символ за валута; в противен случай се приема като число. Валидните символи за валута се определят от свойството **CurrencySymbol** на **NumberFormatInfo**.
AllowHexSpecifier	Указва, че числовият низ представлява шестнайсетична стойност. Валидните шестнайсетични стойности съдържат цифрите 0-9 и шестнайсетичните цифри A-F и a-f. Низовете не трябва да имат "0x" в началото.

Тъй като за всяка настройка се използва един бит от битовото поле, е лесно да се зададе комбинация от настройки. Това можем да направим с побитовия оператор | по следния начин:

```
string price = "3,14159";
double unitPrice = Double.Parse(price,
  NumberStyles.AllowCurrencySymbol |
  NumberStyles.AllowDecimalPoint |
  NumberStyles.AllowExponent |
  NumberStyles.AllowLeadingWhite |
  NumberStyles.AllowParentheses |
  NumberStyles.AllowThousands |
  NumberStyles.AllowTrailingWhite);
```

С този код указваме, че желаем да получим число от потребителски низ, като позволяваме символ за валута, десетична запетая, експонента, празни символи в началото, двойка скоби, разделители на хилядите и празни символи в края. Някои от тези параметри зависят от зададените настройки за култура. По-нататък ще разгледаме влиянието на зададената култура върху метода **Parse(…)**.

За да не се налага всеки път да изброяваме толкова много флагове, можем да използваме предефинираните комбинации от флагове в изброения тип **NumberStyles** – стойностите **Integer**, **Number**, **Float**, **Currency**, **HexNumber** и т. н., имената на които подсказват за какво служат. Ето при-

мер за използването на предефинирана комбинация от флагове при парсването на число:

```
string price = "17,34 лв";
double unitPrice = Double.Parse(price, NumberStyles.Currency);
```

Влияние на културата върху парсването

За да зададем изрично желаните настройки на културата, трябва да използваме варианта **`Parse(string, IFormatProvider)`**. Той приема като параметър обект, имплементиращ **`IFormatProvider`**, който указва форматиращите настройки за използваната култура. **`IFormatProvider`** обектът включва информация за форматирането на числата (разделител за хилядите, знак за дробната част и др.) и за форматирането на валутата (символ за валута и местоположение).

Методът **`Parse(string, NumberStyles, IFormatProvider)`** комбинира предходните два варианта на метода. С този метод може да се зададат едновременно специфичен формат за парсваното число и специфична култура.

Парсване на числови типове – пример

Нека разгледаме няколко примера как можем да парсваме числа като задаваме комбинация от флагове за техния формат и култура:

```
static void Main()
{
  // Set the default formatting for current thread.
  // Invariant culture will be used for Console.WriteLine()
  System.Threading.Thread.CurrentThread.CurrentCulture =
    CultureInfo.InvariantCulture;

  string s = "  000012,54 лв   ";
  double value = Double.Parse(s, NumberStyles.Currency,
    new CultureInfo("bg-BG"));
  Console.WriteLine(value); // 12.54

  s = "17,345,342.38";
  value = Double.Parse(s, NumberStyles.Number,
    new CultureInfo("en-US"));
  Console.WriteLine(value); // 17345342.38

  value = Double.Parse(s, NumberStyles.Number,
    new CultureInfo("bg-BG"));
  // System.FormatException: Input string was not in a
  // correct format.
}
```

При първото обръщение към **Parse(…)** подаваме низ, съдържащ информация за цена в български формат. Задаваме параметър **NumberStyles.Currency**, за да укажем, че това е валута и указваме, че използваме културата за България ("**bg-BG**"), за да може правилно да бъдат разпознати символът за валута ("лв") и знакът за дробната част (",").

При второто обръщение задаваме, че очакваме да прочетем число. Тъй като указваме американска култура ("**en-US**"), коректно се разпознават знакът "," като разделител на хилядите и знакът "." като обозначение за начало на дробната част. За сравнение сме показали, че ако опитаме да прочетем това число с българска култура, ще се получи изключение от тип **FormatException** защото в България символът "," се използва за отделяне на дробната част при реалните числа.

Парсване на дати

Класът **DateTime** имплементира метод **Parse(…)**, който ни позволява да получим нов **DateTime** обект от низове, съдържащи дати в текстов вид.

Нека разгледаме метода **DateTime.Parse(…)** със сигнатура **Parse(string, IFormatProvider, DateTimeStyles)**. Начинът на записване на датата и часа се взимат от **IFormatProvider**. Изброеният тип **DateTimeStyles** задава допълнителни настройки (дали низът може да съдържа празно място, дали да се приема текущата дата, когато е зададен само час и др.).

Освен метода **Parse(…)** класът **DateTime** имплементира и друг метод за парсване на дати и часове – **ParseExact(string date, string format, IFormatProvider)**. Този метод позволява задаване на точния формат на датата и часа, които трябва да се парснат. За разлика от **Parse(…)**, **ParseExact(…)** няма да обработи правилно датата, ако тя не отговаря абсолютно точно на зададения формат.

Форматът на датите и часовете се задава чрез шаблони за форматиране, които са същите, които разгледахме в секцията за форматиране на дати.

Парсване на дати – пример

За да илюстрираме работата с **Parse(…)** и **ParseExact(…)** ще използваме следните примери:

```
DateTime dt = DateTime.Parse("    Thursday,August 05,   2004 ",
  new CultureInfo("en-US"), DateTimeStyles.AllowWhiteSpaces);
Thread.CurrentThread.CurrentCulture =
  CultureInfo.InvariantCulture;
Console.WriteLine("{0:d}", dt);
// Result: 08/05/2004

dt = DateTime.ParseExact("5.08.2004 г. 15:47:00 ч.",
  "d.MM.yyyy г. HH:mm:ss ч.", new CultureInfo("bg-BG"));
Console.WriteLine("{0:d.MM.yyyy HH:mm:ss}", dt);
```

```
// Result: 5.08.2004 15:47:00

dt = DateTime.ParseExact("15:53:21", "HH:mm:ss",
  CultureInfo.InvariantCulture);
Console.WriteLine("{0:HH:mm:ss}", dt);
// Result: 15:53:21
```

При първото обръщение към метода **Parse(…)** указваме, че желаем да конструираме нов обект от тип **DateTime**, като стойността се взима от низа " Thursday,August 05, 2004 ". Вторият параметър създава нов обект **CultureInfo**, който указва използването на настройките за култура на САЩ (US). От изброения тип **DateTimeStyles** се указва стойността **DateTimeStyles.AllowWhiteSpaces**, която разрешава низът, който се парсва, да съдържа интервали, табулации и други символи за празно пространство.

Следващата стъпка е да зададем инвариантна култура за текущата нишка. Както видяхме в секцията за култури, инвариантната култура, не е свързана с конкретно местонахождение. В резултат на тази операция следващите извиквания на **WriteLine(…)** няма да отпечатват датата с настройките на операционната система, а по неутрален относно държавата начин.

За да бъде извършено правилно следващото разпознаване, подаваме на метода **ParseExact(…)** низ с дата, която отговаря символ по символ на шаблона на зададения от нас формат. Задаваме българска култура, за да бъдат разпознати правилно символите за час и година.

При второто обръщение към **ParseExact(…)** подаваме като култура **CultureInfo.InvariantCulture** за да укажем, че низът, който парсваме, не съдържа символи за час, година и т. н.

Кодиращи схеми

При преноса на текстове по мрежата или записа им във файлове се използват т. нар. **кодиращи схеми** (encodings). Най-често се използват кодиращите схеми от стандарта UTF (Unicode Transformation Format), дефинирани от Unicode стандартите.

Кодиращи схеми UTF

UTF представляват алгоритмични кодиращи схеми, при които всеки Unicode номер се представя като уникална поредица от 8-битови, 16-битови или 32-битови стойности. От тук произлизат и различните формати за съхранение на текст в Unicode формат в компютърната памет - UTF-8, UTF-16, UTF-32.

Най-разпространеният в Интернет е UTF-8. UTF-16 се използва в Windows и Java, а UTF-32 е популярен в някои UNIX системи. Конвертирането между различните кодиращи схеми се осъществява от бързи алгоритми, работещи без загуба на информация.

Понеже кодирането при UTF схемите е без загуба, съществува взаимно-еднозначно съответствие между схемите UTF-8, UTF-16 и UTF-32. Това означава, че всеки текст от UTF-8 може да се преобразува до UTF-16 и след това ако се преобразува обратно до UTF-8, ще се получи оригиналният UTF-8 текст без никакви промени. Съществуват и други кодиращи схеми, които работят със загуба, и при тях няма взаимноеднозначно съответствие между оригинален текст и кодиран текст.

UTF-8

При UTF-8 символите се кодират с 8-битови стойности, като символите с номера до 128 (т. е. ASCII символите) се представят със собствения си Unicode номер, като най-старшият им бит е нула. Останалите символи се кодират с два, три или четири байта по определени правила. Това кодиране се използва за предаване на текст, когато е необходим компресиран запис. UTF-8 кодирането е подходящо за текстове, в които преобладават символите от ASCII таблицата, например за текстове на английски език.

При UTF-8 знаците **\u0000** до **\u007F** се записват с един байт. Това са ASCII символите, използвани в САЩ. Знаците **\u0080-\u07FF**, които отговарят на символите от европейските, арабските и еврейските езици, се записват с два байта. Знаците в интервала **\u0800-\uFFFF** се записват с три байта. Всички останали се записват с четири байта.

За източноазиатските езици кодирането UTF-8 е неефективно, защото записва повечето символи с 4 байта.

UTF-16

Кодиращата схема UTF-16 представя един знак (графема) като една или две 16-битови стойности, като по-често използваните Unicode знаци се представят с една 16-битова стойност, а останалите – с две (чрез кодова двойка).

Всички символни низове в .NET Framework инстанциите на класа **System.String**) съхраняват своите стойности в паметта в кодиране UTF-16.

UTF-16 е най-ефективният начин за представяне на азиатски символни низове. При ASCII символите UTF-16 схемата не е много ефективна, защото се използват по два байта за знак, който може да бъде представен само с един байт.

При UTF-16 трябва да се съобрази правилно и подредбата на байтовете, тъй като се срещат два различни подхода при запис. В зависимост дали подредбата на байтовете е **little-endian** или **big-endian** стойността `\uABCD` се записва съответно като `0xCD 0xAB` или като `0xAB 0xCD`.

UTF-32

Както предполага наименованието, в тази кодираща схема се използват 32 бита за всеки Unicode знак. Така всеки един символ се записва като 4-байтово число. В повечето приложения не се налага да използвате нещо повече от ASCII символите и в този случай използването на 32 бита е доста разточително. Затова употребата на UTF-32 е нежелателна при съхранение на диска или при пренос по мрежата.

Подредба на байтовете при UTF-16 и UTF-32

Броят на байтовете не определя еднозначно кодирането на символите. Съществуват различни начини за подреждане на последователността от байтове. Когато първо се записва най-старшият байт, това е big-endian кодиране, а когато той е в края на записа - little-endian. Преди да предадете между два компютъра например една 4-байтова стойност (UTF-32), трябва да се уверите, че те интерпретират по един и същ начин последователността от байтове.

Unicode разрешава проблема с подредбата на байтовете, като дефинира специален символ с номер **`U+FFFE`**, който се нарича "Byte Order Mark (BOM)" и се поставя като идентификатор в началото на Unicode текстовете. Наличието на **`U+FFFE`** в началото на даден UTF-16 файл указва обратен ред на байтове.

Следната таблица показва съдържанието на BOM в различните видове кодиране:

Байтове	Формат на кодиране
00 00 FE FF	UTF-32, big-endian
FF FE 00 00	UTF-32, little-endian
FE FF	UTF-16, big-endian
FF FE	UTF-16, little-endian
EF BB BF	UTF-8

Знаейки че .NET Framework използва кодиране UTF-16 със запис в little-endian формат, можем да проверим коректната стойност на BOM.

Съществуват и други схеми, производни на UTF-16 и UTF-32. Unicode дефинира стандартите UTF-16BE, UTF-16LE, UTF-32LE и UTF-32BE (big-endian, little- endian), които имат гарантирана подредба на байтовете.

Други кодиращи схеми

Освен изброените Unicode кодиращи схеми има и други такива. Повечето от тях кодират Unicode текстовете със загуба. Това означава, че няма взаимоеднозначно съответствие между оригиналния текст и кодирания текст, т. е. от оригиналния текст можем да получим кодирания текст, но обратната операция не е еднозначна.

Например ASCII кодирането преобразува всички Unicode символи с кодове в диапазона `\u0000-\u007F` в съответен ASCII символ, а останалите в символа "`?`" (0x3F). Всички символи от кирилицата при това кодиране се заместват с "`?`" и губят оригиналната си стойност.

Кодиращата схема Windows-1251 замества ASCII символите, символите от кирилицата и някои други с байтове в диапазона 0x00-0xFF, а останалите замества със знак "`?`" (0x3F). Тази кодираща схема често пъти се използва за представяне на текстове на кирилица, но тя не е универсална, както UTF-8 и причинява загуби при някои текстове.

Кодиращи схеми – пример

За илюстрация на разгледаните кодиращи схеми са дадени няколко символа, техните Unicode номера и кодирането им в различните UTF схеми:

Знак	Unicode номер	Unicode номер (hex)	UTF-8 (hex)	UTF-16 (hex)	UTF-32 (hex)
A главна латинска буква "A"	65	U+0041	0x41	0x0041	0x00000041
⅔ математически символ "две трети"	8532	U+2154	0xE2 0x85 0x94	0x2154	0x00002154
♥ символ "сърце"	9829	U+2665	0xE2 0x99 0xA5	0x2665	0x00002665
葉 листо на	33865	U+8449	0xE8 0x91 0x89	0x8449	0x00008449

китайски					
 нота шестнай-сетина	119137	U+1D161	0xF0 0x9D 0x85 0xA1	0xD834 0xDD61	0x0001D161

За да конвертираме текст от символи в байтове и обратно, можем да използваме класа **System.Text.Encoding**.

Конвертиране със System.Text.Encoding

Когато съхраняваме текстова информация на диск или я предаваме по мрежата, тя трябва да бъде представена като поредица от байтове. Когато искаме да прочетем същата тази информация, трябва отново да я преобразуваме в текст. Тези операции са пример за случаи, когато имаме нужда от конвертиране между символи и тяхното байтово представяне в различните разгледани кодиращи схеми.

В .NET Framework преобразуването на текст в последователност от байтове и обратното по дадена кодираща схема можем да извършваме с помощта на класа **System.Text.Encoding** и неговите наследници.

В таблицата са дадени най-често използваните методи и свойства на класа **Encoding**:

Метод/Свойство	Описание
GetBytes(string)	Конвертира символен низ в масив от байтове.
GetString(byte[])	Конвертира масив от байтове в символен низ.
GetChars(byte[])	Конвертира масив от байтове в масив от символи.
GetMaxByteCount(…) **GetMaxCharCount(…)**	Връща максималната дължина на резултата, която може да се получи при конвертиране.
Convert(Encoding, Encoding, byte[])	Конвертира от една схема на кодиране в друга.

Следват няколко примера, които илюстрират използването на тези методи за конвертиране между различните кодиращи схеми:

```
Encoding utf16le = Encoding.Unicode;
byte[] bytes = utf16le.GetBytes("\uABCD");
Console.WriteLine("0x{0:X} 0x{1:X}", bytes[0], bytes[1]);
// Result: 0xCD 0xAB

Encoding utf16be = Encoding.BigEndianUnicode;
```

```
bytes = utf16be.GetBytes("\uABCD");
Console.WriteLine("0x{0:X} 0x{1:X}", bytes[0], bytes[1]);
// Result: 0xAB 0xCD

Encoding windows1251 = Encoding.GetEncoding("windows-1251");
bytes = windows1251.GetBytes("Цакам с А♥ и две ♣.");
Console.WriteLine(BitConverter.ToString(bytes));
// Result: D6-E0-EA-E0-EC-20-F1-20-41-A6-20-E8-20-E4-E2-
// E5-20-A6-2E

Console.WriteLine(windows1251.GetString(bytes));
// Result: Цакам с А¦ и две ¦.
// Windows-1251 encoding is unable to preserve
// some character's original values

Encoding utf8 = Encoding.UTF8;
bytes = utf8.GetBytes("Ω(x) ≤ Ψ(x)π² ≤ ∫φ(x) ∂x ≤ ⅔");
Console.WriteLine(BitConverter.ToString(bytes));
// Result: CE-A9-28-78-29-20-E2-89-A4-20-CE-A8-28-78-
// 29-CF-80-C2-B2-20-E2-89-A4-20-E2-88-AB-CF-86-28-78-29-
// 20-E2-88-82-78-20-E2-89-A4-20-E2-85-94

Encoding ascii = Encoding.ASCII;
bytes = ascii.GetBytes("© 2004 Mente®");
Console.WriteLine(BitConverter.ToString(bytes));
// Result: 3F-20-32-30-30-34-20-4D-65-6E-74-65-3F

Console.WriteLine(ascii.GetString(bytes));
// Result: ? 2004 Mente?
// ASCII encoding is unable to preserve
// some character's original values

Encoding western = Encoding.GetEncoding("iso-8859-1");
bytes = western.GetBytes("© 2004 Mente®");
Console.WriteLine(BitConverter.ToString(bytes));
// Result: A9-20-32-30-30-34-20-4D-65-6E-74-65-AE

Console.WriteLine(western.GetString(bytes));
// Result: c 2004 MenteR
// ISO-8859-1 (latin) encoding is unable to preserve
// some character's original values
```

В разгледания пример първо създаваме обект от тип **Encode.Unicode**. След това извикваме метода му **GetBytes(string)** и запазваме върнатия резултат в масив от байтове. Отпечатваме получения масив от байтове на екрана, като задаваме формат за шестнайсетично число.

Вторият пример е аналогичен но вместо **Unicode** кодираща схема използваме **BigEndianUnicode**, за да укажем, че искаме запис, при който по-старшите байтове се записват преди младшите.

В третия пример създаваме **Encoding** обект с конструктора **GetEncoding()**, като му подаваме като параметър името на кодиращата схема **windows-1251**. Отново с помощта на **GetBytes(string)** взимаме байтовото представяне на низа и с помощта на класа **BitConverter** го отпечатваме на екрана. Класът **BitConverter** е предназначен за преобразуване на базови типове (числа, знаци, низове, булеви стойности) в редици от байтове и обратно. В случая използваме метода му **ToString(byte[])** за да преобразуваме поредицата във вид, удобен за визуализация.

Следващата стъпка демонстрира как да получим обратно символният низ от извлечената поредица от байтове по зададена кодираща схема.

Останалите примери аналогично използват операциите **GetBytes(string)** и **GetString(bytes[])** за **UTF8**, **ASCII** и **ISO-8859-1**. Вижда се, че при някои кодирания преобразуванията са със загуба.

Кодиране Base64

Когато се налага да се пренесат двоични данни през текстова среда, която поддържа само 7-битово кодиране, например при изпращане на ZIP архив по имейл, се налага последователността от байтове да се представи като символен низ. Тази задача се изпълнява най-често с помощта на кодирането Base64.

Кодиращата схема **Base64** преобразува всеки три последователни байта в четири символа измежду **A-Z**, **a-z**, **+** и **/**. В резултат на това кодиране обемът на данните се увеличава с около 1/3.

За да извършим преобразуване на последователност от байтове в **Base64** низ в .NET Framework можем да използваме класа **System.Convert**. Методът **Convert.ToBase64String(byte[])** приема като параметър масива от байтове и по гореописания начин го преобразува в низ.

За преобразуване в обратна посока, от **Base64** низ към масив от байтове, се използва методът **Convert.FromBase64String(base64string)**. Този метод връща като резултат получения масив от байтове. Тази операция се налага да бъде извършвана например когато се записва прикачен към e-mail файл на твърдия диск.

Следният пример показва и двете преобразуващи операции:

```
using System;

class Base64Demo
{
   static void Main()
   {
      byte[] array = {0x00, 0x3D, 0x80, 0xA0, 0xFF};

      string base64 = Convert.ToBase64String(array);
```

```
        Console.WriteLine(base64);
        // Result: AD2AoP8=

        byte[] array2 = Convert.FromBase64String(base64);
        Console.WriteLine(BitConverter.ToString(array2));
        // Result: 00-3D-80-A0-FF
    }
}
```

Работа с Unicode във Visual Studio.NET

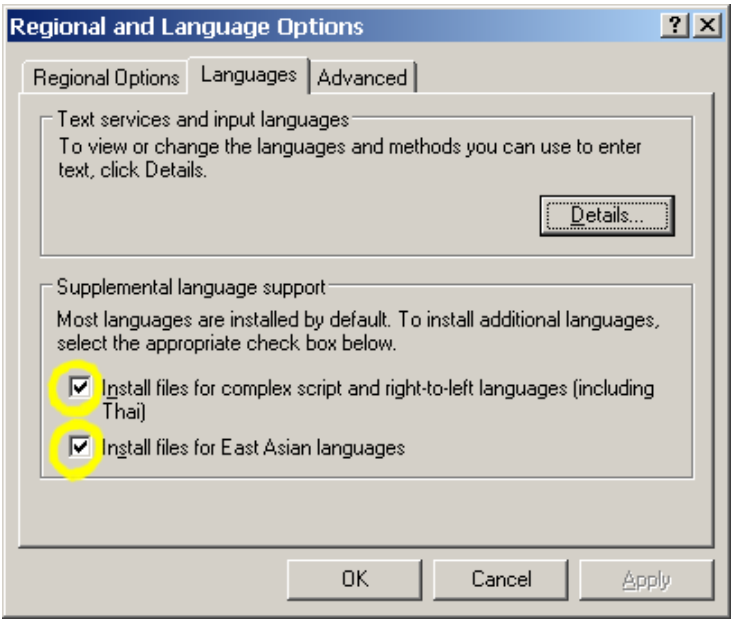
Различните схеми за кодиране за запис на файлове на компютъра важат както за всички текстови файлове така и за файловете със сорс код във Visual Studio.NET. Текстовият редактор има възможност за работа с файлове, записани в голям набор от кодирания.

Ако не укажем изрично каква кодираща схема да се използва, текстовият редактор на Visual Studio.NET използва кодирането по подразбиране в Windows. За да използваме Unicode символи в сорс кода на програмите си, трябва да запишем файловете си с кодираща схема UTF-8.

Поддръжка на всички езици

По подразбиране Windows не поддържа всички езици от Unicode стандарта. С цел спестяване на дисково пространство и други ресурс в Microsoft Windows шрифтовете за източноазиатските езици не се инсталират по подразбиране, но могат да се инсталират ръчно.

Това става като в Control Panel от настройките на Regional and Language Options, както се вижда на фигурата:



Използване на Unicode в редактора на Visual Studio .NET – пример

С настоящия пример ще разгледаме нагледно как чрез редактора на Visual Studio .NET можем да създаваме, компилираме и изпълняваме програми, които съдържат в сорс кода си Unicode низове.

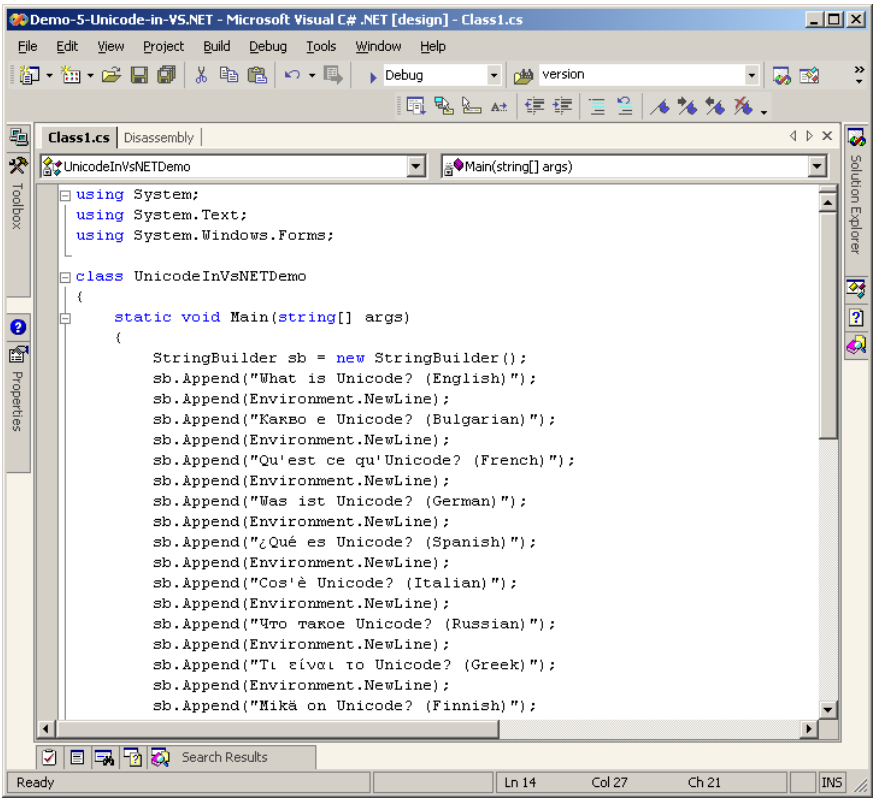
Ето стъпките за изграждането на проект съдържащ Unicode низове:

1. Стартираме VS.NET
2. Създаваме ново конзолно приложение на езика C#
3. Заместваме кода в главния файл (**`Class1.cs`**) със следния код:

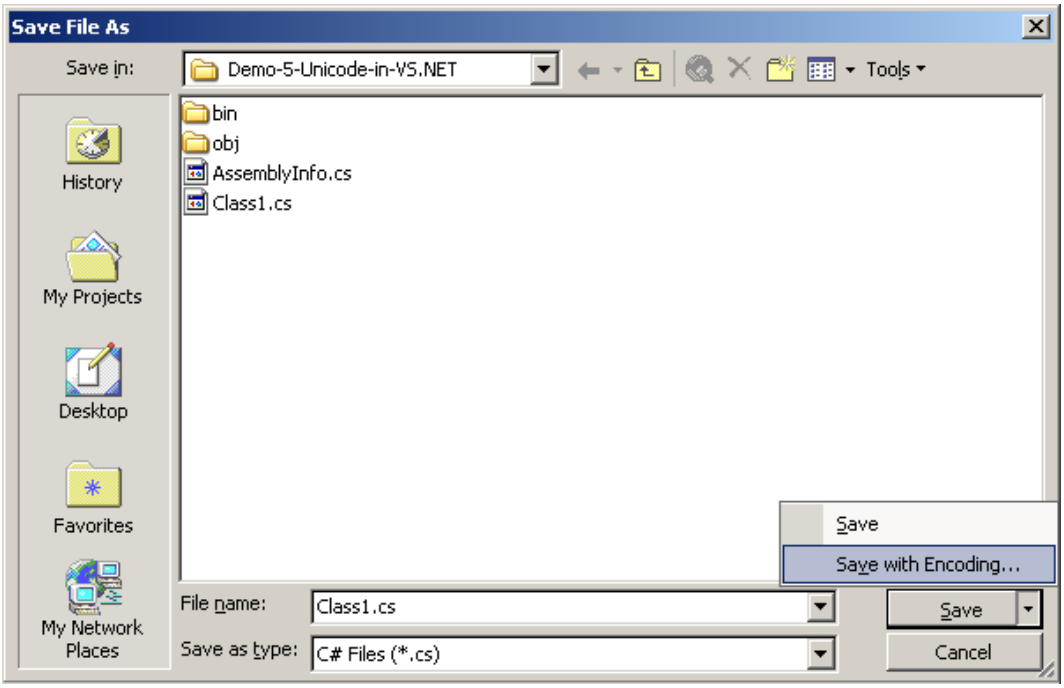
```
using System;
using System.Text;
using System.Windows.Forms;

class UnicodeInVsNETDemo
{
   static void Main()
   {
      StringBuilder sb = new StringBuilder();
      sb.Append("What is Unicode? (English)");
      sb.Append(Environment.NewLine);
      sb.Append("Какво е Unicode? (Bulgarian)");
      sb.Append(Environment.NewLine);
      sb.Append("Qu'est ce qu'Unicode? (French)");
      sb.Append(Environment.NewLine);
      sb.Append("Was ist Unicode? (German)");
      sb.Append(Environment.NewLine);
      sb.Append("¿Qué es Unicode? (Spanish)");
      sb.Append(Environment.NewLine);
      sb.Append("Cos'è Unicode? (Italian)");
      sb.Append(Environment.NewLine);
      sb.Append("Что такое Unicode? (Russian)");
      sb.Append(Environment.NewLine);
      sb.Append("Τι είναι το Unicode? (Greek)");
      sb.Append(Environment.NewLine);
      sb.Append("Mikä on Unicode? (Finnish)");
      sb.Append(Environment.NewLine);
      sb.Append("Što je Unicode? (Croatian)");
      sb.Append(Environment.NewLine);
      sb.Append("Hvað er Unicode? (Icelandic)");
      sb.Append(Environment.NewLine);
      sb.Append("ユニコードとは何か？ (Japanese)");
      sb.Append(Environment.NewLine);
      sb.Append("유니코드에 대해? (Korean)");
      sb.Append(Environment.NewLine);
      sb.Append("什麼是Unicode(統一碼/標準萬國碼)? (Traditional
Chinese)");
```

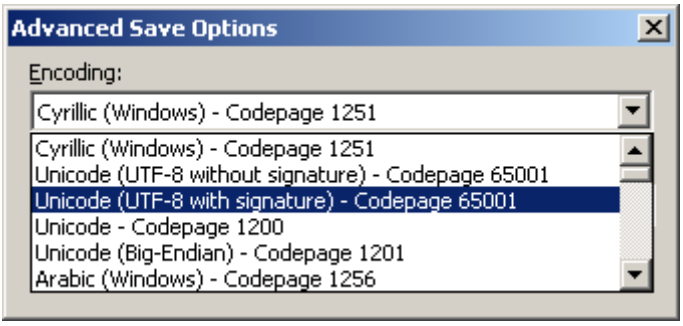
```
        sb.Append(Environment.NewLine);
        sb.Append("什么是Unicode(统一码)? (Simplified Chinese)");
        sb.Append(Environment.NewLine);
        sb.Append(" يونيكود\" (cibarA) ؟");ما هي الشفرة الموحدة\"
        sb.Append(Environment.NewLine);
        sb.Append("რა არის უნიკოდი? (Georgian)");
        sb.Append(Environment.NewLine);
        sb.Append("(werbeH) ?(edocinU) מה זה יוניקוד");
        sb.Append(Environment.NewLine);
        sb.Append("یونی؟چیست کُد  (Persian)");
        sb.Append(Environment.NewLine);
        sb.Append("Unicode คืออะไร? (Thai)");
        sb.Append(Environment.NewLine);
        sb.Append("Unicode là gì? (Vietnamese)");
        sb.Append(Environment.NewLine);
        sb.Append("यूनिकोड क्या है? (Hindi)");
        string s = sb.ToString();
        MessageBox.Show(s, "Unicode Demo");
    }
}
```

Ето как изглежда главният екран на VS.NET:



4. Сега да запишем `Class1.cs` с кодиране UTF-8. Това става по следния начин: **File | Save Class1.cs As... | Save with encoding... | Unicode (UTF-8 with signature) - Codepage 65001.**



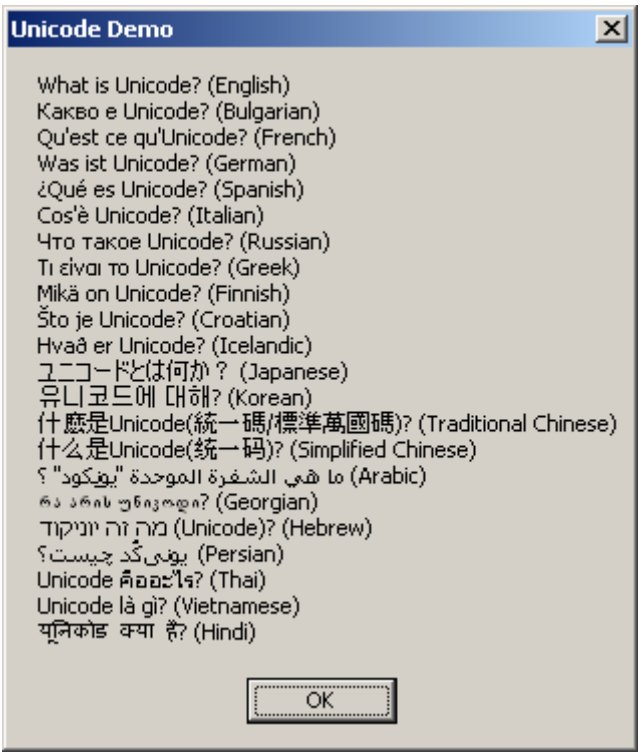


5. Опитваме да компилираме проекта. Получаваме съобщение за грешка, тъй като компилаторът се опитва да намери пространството от имена `System.Windows.Forms` но не може, защото то не е налично по подразбиране в конзолни приложения.

6. Ако компилацията е била неуспешна, добавяме референция в проекта към `System.Windows.Forms.dll`. Това може да стане по следния начин: **Project** | **Add Reference...** | `System.Windows.Forms.dll` | **Select** | **OK**.

7. Компилираме. Компилацията вече минава успешно.

8. Изпълняваме програмата с **[F5]**. На екрана виждаме прозорец със съобщение, в което се извеждат зададените Unicode символи:



9. Ако някои от съобщенията не излизат правилно (например вместо буквите се появяват квадратчета), това се дължи на липсата на поддръжка на някои азиатски шрифтове в текущата инсталация на Windows. Проблемът може да се реши като се инсталират липсващите шрифтове от Regional Options в контролния панел на Windows.

Упражнения

1. Опишете накратко стандарта Unicode. Каква е основната му идея? Има ли връзка между Unicode и типа **`System.Char`** в .NET Framework?
2. Опишете какво представляват символните низове в .NET Framework. Какво е характерно за класа **`System.String`**?
3. Избройте по-важните методи и свойства на класа **`System.String`** и обяснете за какво служат.
4. Напишете програма, която проверява дали в даден числов израз скобите са поставени правилно (има еднакъв брой отварящи и затварящи скоби, които си съответстват). Използвайте методите и свойствата на класа **`System.String`**.
5. Напишете програма, която по дадена последователност от символи (цел) и даден текст извлича от текста всички думи, които съдържат

зададената цел в себе си като подниз. Използвайте само класа **System.String**.

6. Напишете програма, която по даден URL адрес във формат **[protocol]://[server]/[resource]** извлича от него отделните му елементи – **[protocol]**, **[server]** и **[resource]**. Например за URL **http://www.devbg.org/forum/index.php** трябва да извлече **[protocol]** = "**http**", **[server]** = "**www.devbg.org**" и **[resource]** = "**/forum/index.php**". Използвайте методите и свойствата на класа **System.String**.

7. Напишете подпрограма, която заменя в даден низ всички поднизове, оградени с таговете **<upcase>** и **</upcase>** с еквивалентни на тях низове с главни букви. Подпрограмата трябва да се справя с вложени тагове и (доколкото е възможно) с некоректно зададени тагове. Например при вход:

```
проба <upcase>алабала</upcase> </upcase>хахо<upcase>тест
<upcase> и пак</upcase> аха</upcase> eho
```

8. трябва да връща резултат

```
проба АЛАБАЛА хахо ТЕСТ И ПАК АХА eho
```

9. Напишете програма, която обръща думите в дадено изречение в обратен ред. Например изречението "Брала мома сладки къпини" трябва да се преобразува в "Къпини сладки мома брала". Използвайте класа **StringBuilder**.

10. Напишете програма, която претърсва даден текст за дадена дума и намира и отпечатва всички изречения, в които тази дума се среща. Можете да считате, че границата между две изречения е някой от символите ".", "!" и "?", следван евентуално от празно пространство и след него от дума, започваща с главна буква на кирилица или латиница. Например в текста "\tНалей ми бира! Изстина бирата заради тези символни низове. Ще сложа две-три в камерата.\n \t Отивам до магазина за още бира." думата "бира" се среща само в първото и последното изречение. Използвайте методите и свойствата на класа **System.String**.

11. Даден е речник с думи, който представлява текст във формат "дума значение" – по една речникова единица на всеки ред. Да се състави програма, която по дадена дума намира значението ѝ в речника. Използвайте методите и свойствата на класа **System.String**.

12. Дадени са текст и списък от думи, разделени със запетайка. Списъкът описва думи, които са нецензурни и не трябва да се съдържат в текста. Напишете програма, която замества всички нецензурни думи от текста със звездички. Например ако имаме текста "Какъв хикс дириш тука бе менте?" и списък с нецензурни думички "менте, хикс", програ-

мата трябва да го преработи така: "Какъв **** дириш тука бе *****?". Използвайте класовете **String** и **StringBuilder**.

13. Напишете програма, която заменя в даден HTML документ всички хипервръзки **…** с метаописание на тези връзки във формат **[url href=…]…[/url]**. Програмата трябва да се справя с вложени тагове и дори с вложени хипервръзки (въпреки че това не е позволено в езика HTML). Използвайте методите и свойствата на класовете **String** и **StringBuilder**.

14. Напишете програма, която изважда от даден текстов документ всички поднизове, които приличат на email адрес (последователности от символи във формат **<identifier>@<host>…<domain>**). Използвайте методите и свойствата на класа **System.String**.

15. Напишете метод, който приема като вход символен низ и го отпечатва във вид на последователност от байтове в шестнайсетична бройна система във формата, в който се дефинират низове в C#. Например за низа "Hi!" трябва да се отпечата "\x48\x69\x21". Използвайте методите и свойствата на класа **System.String** и подходящи форматиращи низове.

16. Напишете програма, която очаква въвеждане на дата в някой от форматите "dd.mm.yyyy", "dd/mm/yyyy", "dd.mm.yy" или "dd/mm/yy", парсва въведения текст и при успешно разпознаване на някой от форматите отпечатва датата във формата по подразбиране за текущата култура.

17. Напишете програма, която за всяко цяло число от 5 до 25 отпечатва таблица с 4 колони – числото (подравнено отдясно, разположено в пространство от 5 символа), числото на квадрат (подравнено вляво, разположено в пространство от 6 символа), корен квадратен от числото (с точност 4 десетични цифри, разположен в пространство 10 символа, допълнено с нули в началото) и натурален логаритъм от числото (разположен в пространство от 8 символа, с възможно най-голяма точност, така че да не остава празно място). Използвайте доколкото е възможно форматиращи низове.

18. Напишете програма, която въвежда от конзолата едно реално число и го отпечатва във вид на валута, специфична за държавите: Австрия, България, Израел, Монголия, Тайван и Япония. Ако на конзолата някои символи не излизат правилно, използвайте метода **MessageBox.Show()**.

19. Реализирайте клас "обикновена дроб", който съдържа числител и знаменател (без операциите над обикновени дроби) и му дефинирайте необходимите методи за потребителско форматиране, така че да може да се отпечатва в подходящ вид с **Console.WriteLine**. Дефинирайте и два форматиращи низа ("N" и "R"), които отпечатват дробта като обикновена дроб (напр. "1/4") и като реално число с точност 2 знака (напр. "0,25").

20. Напишете програма, която въвежда символен низ от клавиатурата и се опитва да го преобразува в число, използвайки формати "XXX XXX XXX.YY", "XXX XXX XXX,YY", "XXXXXXXXX,YY" и "XXXXXXXXX.YY". Програмата трябва да отпечатва за всеки един от тези формати дали парсването е било успешно и какъв резултат се е получил в случай на успех. Използвайте метода **`Double.Parse`** и свойствата за разделители на числа от класа **`CultureInfo.NumberFormat`**.

21. Напишете програма, която прочита символен низ от конзолата, преобразува го в различни формати (UTF-8, ASCII, windows-1251) и го отпечатва като последователност от байтове в шестнайсетичен вид. Използвайте класа **`System.Text.Encoding`**.

Използвана литература

1. Светлин Наков, Символни низове (strings) – http://www.nakov.com/dotnet/lectures/Lecture-8-Strings-v1.0.ppt
2. MSDN Library – http://msdn.microsoft.com
3. Jeffrey Richter, Applied Microsoft .NET Framework Programming, Microsoft Press, 2002, ISBN 0735614229
4. Charles Petzold, Programming Windows with C#, Microsoft Press, 2003, ISBN 954-685-239-2
5. Tom Archer and Andrew Whitechapel, Inside C#, Microsoft Press; 2nd edition 2002, ISBN 0735616485
6. MSDN Training, Programming with the Microsoft®. NET Framework (MOC 2349B), Module 7: Strings, Arrays, and Collections
7. Stephen Gilbert, Computer Science 140, C# Programming, Strings and Exceptions – http://csjava.occ.cccd.edu/~gilberts/CS140S03/slides/140S0309.ppt

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 10. Регулярни изрази

Необходими знания

- Базови познания за общата система от типове в .NET
- Базови познания за езика C#
- Познания на средствата за работа със символни низове в .NET Framework

Съдържание

- Регулярни изрази – същност, произход и основни приложения
- Прости примери за регулярни изрази
- Езикът на регулярните изрази. Основни елементи на синтаксиса
- Регулярните изрази в .NET платформата. Пространството **`System.Text.RegularExpressions`**.
- Търсене с регулярни изрази
- Работа с групи
- Валидация с регулярни изрази
- Заместване и разделяне по регулярен израз
- Настройки и опции
- Предварително компилиране на регулярни изрази
- Ефективност на регулярните изрази
- Полезни готови регулярни изрази
- Интернет ресурси. Инструментът The Regulator

В тази тема ...

В настоящата тема ще разгледаме регулярните изрази, набиращи все по-голяма популярност сред разработчиците на софтуер при решаването на проблеми, свързани с обработката на текст. Ще се спрем на произхода и същността на регулярните изрази и ще се запознаем със синтаксиса и основните правила при конструирането им. Темата предлага кратко представяне на основните дейности, при които е подходящо използването на регулярни изрази, и дава конкретни насоки как можем да правим това със средствата на .NET Framework. Ще разгледаме инструментариума, който библиотеката с класове предоставя за работа с регулярни изрази, и ще опишем най-важните методи, съпроводени с достатъчно примери.

Регулярни изрази

Регулярните изрази представляват много мощен апарат за обработка на символни низове. Множество често срещани в практиката задачи, могат да бъдат решени изключително просто и елегантно с тяхна помощ. В .NET платформата регулярните изрази са широко функционални и същевременно лесни за използване. Ето защо доброто познаване на средствата за работа с тях може да спести много главоболия и трудности, познати на всеки, който се занимава с обработка на символни низове.

Какво е регулярен израз?

Регулярният израз е символен низ, конструиран по специални синтактични правила. Той описва един език от думи – символни низове. За всяка дума е дефинирано еднозначно дали принадлежи на описвания език, или не. На регулярния израз може да се гледа като на шаблон – той търси съвпадения с всички думи, които отговарят на този шаблон. Един от най-често използваните регулярни изрази, например, е изразът:

```
(.+)@(.+)\.(.+)
```

Макар да изглежда странно на пръв поглед, това не е нищо друго освен един опростен шаблон за формата на стандартните e-mail адреси, с които всеки е запознат – поредица от символи, следвана от символа `@` и после име на домейн. Как точно регулярният израз представя този шаблон, ще разберем в настоящата тема.

За какво се използват регулярните изрази?

Регулярните изрази могат да се използват за най-разнообразни задачи при текстообработката. В практиката те най-често са полезни при три типа задачи – търсене, валидация и заместване.

Търсене и извличане на информация

Търсенето на даден низ в текст е полезно при редица проблеми от практиката. Принципно то може да се реализира и без помощта на регулярни изрази, но те значително го улесняват и позволяват паралелното извличане на различна информация от текста – поднизове, които носят определен смисъл за нас. Класическите методи за обхождане и парсване на текста обикновено изискват значително повече време и количество код в сравнение с решенията, използващи регулярни изрази.

Валидация на входни данни

Валидацията на входни данни е задължителна във всеки съвременен софтуер, който претендира да спазва елементарните изисквания за сигурност. С помощта на регулярни изрази тя се реализира изключително

лесно – възможни са най-различни видове валидация на символните данни, в зависимост от нуждите на програмиста.

Заместване

Заместването на един низ с друг в текст е една от най-често срещаните практически задачи при текстообработката, а същевременно и доста бавна и трудна за реализация. С помощта на регулярните изрази подобен проблем се разрешава само с няколко реда код. При това имаме възможност да заместваме по шаблон, а не само с фиксиран текст, като в шаблона можем да използваме и елементи от търсения низ.

Например можем да намерим всички срещания на конструкции от типа "(a+b)*c" и да ги заместим с еквивалентните им "a*c+b*c" – нещо, което е доста трудно за постигане с традиционните средства за синтактичен анализ и текстообработка.

Регулярни изрази и крайни автомати

Регулярните изрази са разработени първоначално като математическа теория. Те са свързани с разпознаването и обработката на т.нар. формални езици – клон от дискретната математиката, свързан със строги синтактични дефиниции на абстрактни обобщения на реалните човешки езици. Теорията на формалните езици днес намира широко приложение в различни области от математиката, лингвистиката и компютърните науки, например в изкуствения интелект, търсенето в Интернет, предпазването от спам и др.

Какво са крайни автомати?

Един **регулярен израз** описва един **регулярен (автоматен) език.** В теорията това са езиците с най-строги граматични правила, които могат да се обработват от машини. На всеки **регулярен език** отговаря еднозначно т. нар. **краен автомат** – абстрактна математическа машина, която лесно се реализира програмно и представлява набор от състояния и правила за преход между тях. Проверката дали един текст представлява дума от регулярния език, описван от даден регулярен израз, практически се свежда до реализирането на съответния краен автомат.

Понеже работата на крайните автомати е бърза, регулярните изрази са доста ефективен метод за обработка на текст. За пълната им функционалност, реализирана в съвременните програмни езици, обаче, се налага известно разширение на крайните автомати, което намалява ефективността, макар да повишава значително възможностите на обработката. На това ще се спрем по-късно.

История на регулярните изрази

Както вече споменахме, регулярните изрази първоначално са възникнали като математически апарат за описание на регулярните езици в изчислителната теория и теорията на формалните езици. С навлизането на ком-

пютрите в употреба при научните изследвания, регулярните изрази и крайните автомати се имплементират в програмните среди и операционните системи, главно с цел обработка на текст. С течение на времето възможностите на регулярните изрази в езиците за програмиране се разширяват значително, до степен, в която те реално вече не представят само ограничените регулярни езици, а и по-широки категории формални езици.

Регулярните изрази в езиците за програмиране

Регулярните изрази са въведени най-напред в Unix и различните програми и команди за Unix, например **ed**, **grep**, **awk**, **lex**, **Emacs** и др. По-късно е разработена библиотека за C, наречена "regex", която обаче е сравнително сложна и неудобна за употреба. Истинският бум на този апарат се дължи на езика Perl, който вгражда в самия си синтаксис много удобни средства за работа с регулярни изрази и улеснява значително задачите за текстообработка, на които вече се спряхме. Впоследствие Филип Хейзъл разработва "pcre", което наподобява по функционалност възможностите, разработени в Perl, и се използва в по-новите езици Python и PHP, а библиотека за регулярни изрази е реализирана и в Java.

В .NET платформата с много малки разлики се използват принципите, въведени от най-разпространения Perl синтаксис на регулярните изрази, като класовете, предоставяни от библиотеката, предоставят всичко необходимо за ефективна работа с тях.

Няколко прости примера

Както вече обяснихме, регулярните изрази сами по себе си са низове, които използваме, за да търсим съвпадения на символни последователности с дефинирания шаблон. Да разгледаме няколко съвсем прости примера, които ще ни помогнат по-лесно да се ориентираме в основните възможности на регулярните изрази.

Лесен пример за начало

Следният низ е един прост регулярен израз:

```
пример номер [0-9]+
```

Този шаблон ще намери съвпадения с всички поднизове, които започват с фразата "пример номер", следвана от поне една десетична цифра.

Квадратните скоби в езика на регулярните изрази указват клас от символи – в случая това са всички символи в диапазона между 0 и 9. Знакът **+** ни казва, че предходният символ (или в случая целият клас символи **[0-9]**) трябва да се среща последователно 1 или повече пъти. Така низът "пример номер 987" ще бъде счетен за съвпадение с шаблона, както и низът "пример номер 05". Низът "пример номер" няма да е съвпадение, защото по шаблона трябва да има поне една цифра на края.

С този шаблон ние можем, разбира се, да търсим съвпадения и в по-дълъг текст. Да разгледаме например низа "Това е вашият пример номер 10". В този низ има подниз, който съвпада с шаблона и той се счита за съвпаднат по регулярния израз – това е поднизът "пример номер 10".

Търсене на телефонни номера

Да разгледаме друг пример:

```
(+359|0)88[0-9]{7}
```

С този израз можем да намерим съвпадения с номера на мобилни телефони в мрежата на М-Тел. Отново използваме символния клас за цифри `[0-9]`, следван от модификатора за количество `{7}` – това означава, че трябва да има точно 7 повторения на символи от класа `[0-9]`. В началото имаме `(+359|0)`, което означава, че номерът трябва да започва или с `+359`, или с `0`. Символът `|` обозначава алтернативни възможности. Примери за номера, който съвпадат с шаблона, са "`+359887675340`" и "`0888997621`", но не и "`188385953`" (започва с 1), или "`+3598890076`" (има само пет цифри след "`+35988`").

Пример за регулярен израз в .NET

Преди да преминем към по-детайлното описание на езика на регулярните изрази, нека разгледаме и един цялостен пример за използване на такъв израз в .NET:

```
static void Main()
{
   string text = "Няма скара, няма бира, няма какво да ям";
   string pattern = @"\w*ира|скара|\w*ям\w*";
   Match match = Regex.Match(text, pattern);
   while (match.Success)
   {
      Console.WriteLine(
         "Низ: \"{0}\" - начало {1}, дължина {2}",
         match, match.Index, match.Length);
      match = match.NextMatch();
   }
}
```

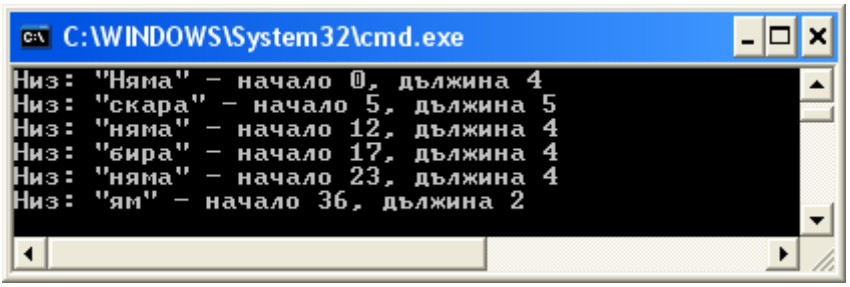
Как работи примерът?

Регулярният израз, който се съдържа в низа **`pattern`**, търси за три алтернативни възможности:

- подниз, завършващ на "**`ира`**" – това се сочи от частта "**`\w*ира`**", където **`\w*`** означава произволна последователност (може и с нулева дължина) от букви, цифри и знак за подчертаване.

- подниз "**скара**".
- подниз (отново от букви, цифри и знак за подчертаване), който съдържа подниза "**ям**".

Както виждаме, за търсенето на съвпадения се използват методите на класовете **Regex** и **Match**. Те ще бъдат разгледани подробно по-късно в темата. Изходът от горния пример е следният:



Езикът на регулярните изрази

Богатството на синтаксиса на регулярните изрази позволява голяма гъвкавост при изграждането на шаблони. Пълните възможности на езика рядко се ползват в практиката, но за по-специфични задачи все пак може да се наложи и тяхното познаване. Ние ще разгледаме първо с по-основните и често използвани конструкции, а по-нататък в темата ще обясним и някои от по-интересните детайли на регулярните изрази.

В синтактичен план езикът на регулярните изрази се дели на две основни групи – **литерали** и **метасимволи** (символи със специално значение).

Литерали

Литералите са символи и поднизове в регулярния израз, които се възприемат от апарата за обработка буквално, в стандартния си смисъл. Литералите се използват, за да фиксираме частта от шаблона, която задължително трябва да присъства точно в посочения вид. Например в регулярния израз, съхраняван в променливата **pattern**, която разгледахме в примера по-горе, частите "**ира**", "**скара**" и "**ям**" са литерали – те се търсят в текста от **машината на регулярните изрази** (**regex engine**) точно в този си вид, в който ги виждаме написани.

Метасимволи

Метасимволите са група от символи, които имат специално значение в езика на регулярните изрази. С тяхна помощ реално се постига цялата функционалност и описателност на апарата, който разглеждаме. Метасимволите, с които разполагаме, осигуряват достатъчно гъвкавост, за да можем да опишем практически почти всеки шаблон, който би могъл да ни трябва.

Символите, които се третират по специален начин от машината на регулярните изрази в зависимост от позицията си, са следните: `[`, `]`, `|`, `*`, `.`, `+`, `?`, `(`, `)`, `{`, `}`, `\`, `^` и `$`. Най-често обаче терминът "метасимволи" се използва не за тях, а за специалните конструкции, които носят определен смисъл за шаблона и в които тези символи участват.

В примера, който разгледахме по-горе, метасимволи са "`\w`" и "`*`", както и "`|`". Вече обяснихме донякъде тяхното значение, а останалите специални символи ще разгледаме малко по-нататък.

Escaping при регулярните изрази

Естествено, тъй като метасимволите се третират по нетривиален начин при парсването на регулярния израз, те не могат да се използват като литерали директно и не могат да се търсят като обикновени символи. Както и при повечето програмни езици, този проблем се избягва с помощта на escaping чрез символа `\` (backslash). Поставен пред някой от специалните символи, които изброихме по-горе, този символ отменя особеното им значение и те се третират по обикновения начин. Например `\*` означава `*`, `\+` означава `+` и т.н. Това важи разбира се и за самия символ `\`, т.е. ако искаме да го използваме като литерал, трябва да използваме `\\`.

Ще обърнем внимание, че някои от изброените по-горе метасимволи имат специално значение само на определени места в шаблона. Например символът `]`, както и символите `)` и `}` са метасимволи само ако преди тях в шаблона има незатворена съответната отваряща скоба. Ако се опитаме да ползваме escaping на тези символи в друг контекст, ще получим грешка, защото това обърква парсването.

Всъщност всички срещания на символа `\`, последван от символ, който не е разпознат като специална escaping последователност, водят до изключение по време на изпълнението на програмата.

Други escaping последователности

Както при повечето езици за програмиране, в регулярните изрази може да се използват познатите последователности за обозначаване на специалните ASCII символи, за които няма видимо означение. Ще изброим някои от тях:

- `\n` – нов ред (new line)
- `\r` – връщане на каретката (carriage return)
- `\t` – табулация (tab)
- `\a` – звуков сигнал (bell)
- `\b` – връщане назад (backspace)

Комбинацията `\b`, както ще видим по-нататък, всъщност има по-специално значение за синтаксиса на регулярните изрази. Ето защо можем да я използваме като backspace единствено в споменатата конструкция за клас

от символи **[]** и в шаблоните за заместване, за които ще стане дума по-късно.

Escaping последователности за Unicode и ASCII кодове

Езикът на регулярните изрази дава възможност да се използват директно ASCII и Unicode кодове на символи за означението им в шаблона. Това става по три начина:

- **\XXX** – символът ****, последван от най-много три цифри, представя ASCII символ, въведен с осмичен код. Например **\040** е интервал, а **\081** – буквата **A**. Позволено е и даже се препоръчва да се използват водещи нули, за да се избягва двусмислие с подобната конструкция за обратни препратки, за която ще говорим по-късно.
- **\xXX** – комбинацията **\x**, последвана от точно две цифри, обозначава ASCII символ в шестнайсетичен код. Например интервалът е **\x20**, а буквата **A** – **\x41**.
- **\uXXXX** – тази комбинация (с точно четири цифри) представя Unicode символен код също чрез шестнайсетични числа. В този формат символът за интервал се означава като **\u0020**, а буквата **A** – като **\u0041**. Символът **ï**, например, можем да означим с **\u00EF**.

Някои особености

Когато използваме escaping при регулярните изрази в .NET Framework, не трябва да забравяме, че се налага да се съобразяваме и с компилатора. Преди да се подадат на машината на регулярните изрази, низовете, които използваме за шаблони, първо се обработват от парсер, който има собствени escaping последователности.

В езика C# символът **** се използва в низовете също за escaping. Трябва да се съобразяваме с този факт, защото можем да стигнем до нежелани резултати, ако не внимаваме. Да разгледаме шаблона "**\w***", който както ще видим по-късно, най-общо означава произволна дума. Ако напишем този низ в нашия сорс код, ще получим грешка при компилация, защото компилаторът на C# не може да разпознае "**\w**" като коректна escaping последователност. Ето защо трябва да освободим символа **** от специалното му значение, като използваме escaping и за него – "**\\w***". Сега вече след преминаването през парсера на C# компилатора този низ ще има вида "**\w***", което е коректно за машината за регулярни изрази, която ще го обработи по време на изпълнение.

Какво трябва да направим, ако искаме да използваме **** като обикновен символ в израза (когато търсим последователност, в която той участва)? Тогава трябва да имаме "****" в шаблона, например "**c:\\windows**". Но преди да се разчете като шаблон от машината за регулярни изрази, този низ ще мине през парсера по време на компилация и там двата символа backslash ще се редуцират до един по правилата за escaping в C#, при което ще получим в резултат "**c:\windows**". Тук обаче **\w** има значение на

метасимвол и ще получим нежелан резултат. Затова в сорс кода трябва да имаме "**c:\\\\windows**" – т.е. четири повторения на \, за да намерим съвпадение с един такъв символ.

Използване на символа @

Както отбелязахме в темата за символните низове, в езика C# при поставянето на @ пред отварящите кавички на низа всички специални символи в него (освен кавичките) губят специалното си значение. Това е много удобно при шаблоните за регулярни изрази и е препоръчително да се използва винаги, защото спестява проблемите, описани в горния абзац. Горните два примера можем да променим така:

```
string patternWord = @"\w+";
//Same as: patternWord = "\\w+";
string patternDir = @"c:\\windows";
//Same as: patternDir = "c:\\\\windows";
```

Използването на @ ни позволява да записваме низа във вида, в който той ще се прочете от машината за регулярните изрази (с малки изключения за кавичките). Това е доста удобно и е добре да го използваме.

Най-важното за работата с регулярни изрази

Преди да преминем към разглеждането на по-специалните конструкции в синтаксиса на регулярните изрази, е добре да знаем малко повече за начина, по който се откриват съвпадения с шаблона. Следните прости правила, които трябва да запомним, обясняват накратко как машината на регулярните изрази търси за такива съвпадения.

Шаблонът не е за съвпадение с целия низ

Когато търсим съвпадение по даден шаблон, ние търсим произволен подниз на нашия низ, който да отговаря на шаблона. В текста може да има много такива поднизове. Шаблонът не описва целия текст (освен ако изрично не укажем това), а описва поредица символи, която се търси в текста. Например шаблонът "**бира**" има две съвпадения в низа "**Разбирам от бира**" – едното е част от думата "**разбирам**", а другото е самата дума "**бира**".

Съвпаденията се откриват в реда на срещане

Регулярните изрази винаги работят отляво надясно по низа и откриват първо най-левите съвпадения. В горния пример поднизът "**бира**" в "**Разбирам**" ще бъде открит пръв, макар че ние вероятно сме искали да търсим за самата дума "**бира**". Това е често срещан проблем и по-нататък в темата ще се научим как да се справяме с него.

Търсенето приключва, когато се открие съвпадение

По принцип целта на търсенето с регулярни изрази е да се открие някакво съвпадение с шаблона. Машината на регулярните изрази се стреми да не хаби излишни ресурси и прекратява хода си в момента, в който се открие съвпадение. Ако полученото съвпадение не ни устройва, можем да търсим отново или да използваме подходящите методи за събиране на всички съвпадения до пълното изчерпване на низа наведнъж.

Търсенето продължава от последното съвпадение

Ако сме намерили съвпадение с регулярен израз и след това подновим търсенето в същия текст със същия регулярен израз, то продължава от мястото след последното намерено съвпадение.

Това е важна подробност, която трябва добре да се запомни, защото често е причина за объркване! За да я изясним, ще разгледаме примерния шаблон "**(бира!){2}**". С този шаблон търсим две последователни повторения на подниза "**бира!**", тоест на практика търсим "**бира!бира!**". Нека да го приложим към примерния низ "**бира!бира!бира!**". В текста реално има две срещания на нашия шаблон – едното започва от позиция 0 ("**бира!бира!**бира!"), а другото – от позиция 5 ("бира!**бира!бира!**").

С търсенето обаче не можем да получим и двата резултата. При първия опит ще получим по-левия подниз (този от позиция 0). Ако сега търсим отново, то новото търсене започва от позиция 10 (това е първата позиция след края на намереното съвпадение), а оттам до края на низа вече има само едно "**бира!**" и няма да имаме ново съвпадение.

Регулярният израз търси за всички възможности подред

Това е основният принцип, който описва работата на машината за регулярните изрази. При всеки следващ символ тя опитва подред всички възможности за намиране на съвпадение от тази позиция и връща първото успешно или продължава нататък. Понякога това може да доведе до търсене с връщане назад (backtracking), за което ще стане дума по-късно.

Основни метасимволи

Сега ще разгледаме най-често срещаните метасимволи в езика на регулярните изрази. За човек, който никога не се е сблъсквал с подобна материя, долните редове може да изглеждат неразбираеми на първо четене. Всъщност обаче е необходима единствено малко практика и разучаване на примерите, за да се овладее боравенето с регулярните изрази. Впоследствие следващите редове могат да служат и за справочник, към който да се прибягва в случай на нужда.

Класове от символи

Това са метасимволи, които означават цяло множество от обикновени символи. Срещането на който и да е символ от множеството (или класа) на съответното място в низа се счита за съвпадение. Използват се следните означения за различните видове класове от символи:

Точка

Специалният символ . обозначава класа от всички символи, с изключение на символа **\n** за нов ред (в Windows това е **\r\n**). Ако е включена опцията **Singleline**, точката обозначава и символа за нов ред (за опции ще стане дума по-късно в темата). Като пример за този метасимвол, да разгледаме шаблона "**би.а**" и няколко низа, към които да го приложим:

Шаблон: **би.а**
бира – има съвпадение "бира"
бива – има съвпадение "бива"
би+а – има съвпадение "би+а"
бивна – няма съвпадение (между "би" и "а" има два символа, а не един)

Квадратни скоби (класове с изброяване)

Конструкцията **[редица_символи]** е метасимвол, обозначаващ класа от всички обикновени символи, изброени в редицата. Символите се изброяват без никакви разделители между тях. Тук има няколко варианта. Единият е простото изброяване на символи от вида **[символи]**. Това е стандартният вид на конструкцията, който вече обяснихме:

Шаблон: **би[вр]а**
разбира – има съвпадение "бира"
не бива – има съвпадение "бива"
биха ни – няма съвпадение (между "би" и "а" трябва "р" или "в")

Обърнете внимание, че редът на символите в квадратните скоби значение за реда на намерените съвпадения – важен е редът на срещане в низа. Съвпадението с "**бира**" е първо в "**Тази бира я бива**", въпреки че в шаблона "**в**" стои преди "**р**" в квадратните скоби.

Отрицание на клас

Това е конструкцията **[^символи]**. Символът **^**, поставен веднага след отварящата квадратна скоба, означава, че се търсят съвпадения с всички символи, които **НЕ** влизат в класа на изброяваните. Важно е да се запомни, че символът **^** има специално значение, единствено когато е поставен веднага след отварящата квадратна скоба, иначе се третира като обикновен символ:

Шаблон: **би[^вр]а**
разбира – няма съвпадение (между "би" и "а" не трябва да има "р" и "в")
биха ни – има съвпадение "биха"

Шаблон: **би[в^р]а**
разбира – има съвпадение "бира"
биха ни – няма съвпадение (между "би" и "а" трябва "р", "^" или "в")
би^а – има съвпадение "би^а"

Изброяване с диапазон

Можем да използваме и следния вид клас от символи: **[символА-символВ]**. Този вариант на конструкцията с квадратните скоби търси съвпадения с всички символи, намиращи се в затворения интервал между **символА** и **символВ** в кодовата таблица. Подобно на **^** символът **–** има специално значение, само ако е между други символи, а не в началото и в края.

Шаблон: **би[б-п]а**
разбира – няма съвпадение ("р" не е в интервала "б-п")
не бива – има съвпадение "бива"

Шаблон: **би[-бп]а**
не бива – няма съвпадение (между "би" и "а" трябва "б", "п" или "-")
би-а – има съвпадение "би-а"

Ще отбележим, че трите вида изброяване могат свободно да се комбинират – например "**[^a-zA-Z01]**" намира съвпадение с всеки символ, който не е малка или голяма латинска буква, нито цифрите **0** или **1**.

Специалните символи в конструкцията за клас

Важно е също да отбележим, че с няколко изключения всички метасимволи губят смисъла си на метасимволи, ако се използват при изброяване на клас от символи. Например символът точка в израза "**te[s.]t**" се приема за литерал и този шаблон се удовлетворява от "**test**" и "**te.t**", но не и от "**teat**" или какъвто и да е подобен низ с трети символ, различен от **.** и **s**. Изключенията са познатите вече символи:

- **^** има специално значение след символа **[**.
- **–** има специално значение навсякъде освен след **[** или **[^** и преди **]**.
- **]** има специално значение (край на изброяването) навсякъде освен след **[** или **[^** (празни квадратни скоби водят до грешка).
- **** винаги има специално значение на escaping character и за да го използваме като литерал, ни трябва ****.

Предефинирани класове от символи

Няколко специални метасимвола обозначават предварително фиксирани класове, чиято употреба често пъти е удобна. Ще ги разгледаме един по един, а като общо правило може да се запомни, че метасимволите за отрицание (т.е. непринадлежност към даден предефиниран клас) са същите като съответните за принадлежност към дадения клас, но с главна буква:

- **`\w`** – Означава всички т. нар. "**alphanumeric characters**" – букви, цифри и знака за подчертаване. В общия случай (с употреба на Unicode), това включва букви и цифри от всички използвани в Unicode азбуки. Ако е включена опцията **ECMAScript**, **`\w`** е еквивалентен само на **`[A-Za-z_0-9]`**.
- **`\W`** – Означава всички символи, които не принадлежат на горния клас.
- **`\s`** – Означава всички "**whitespace characters**", т.е. празно пространство – интервали, табулации, символа за нов ред и т.н.
- **`\S`** – Означава всички символи, които не са празно пространство.
- **`\d`** – Означава всички десетични цифри (от Unicode езиците).
- **`\D`** – Всички символи, които не са десетични цифри.

Особено внимание при тези метасимволи трябва да се отдели на символа `\` пред тях, което налага някои особености при escaping в C#, за които вече споменахме. Често срещани грешки стават и при използването на предефинираните класове в конструкцията с квадратните скоби. Това може да се види и в примерите, които следват:

Шаблон: **`\d\w\w\s\w\w\w\w\W`**
`4та бира!` – има съвпадение "4та бира!"
`5те пици, 3те тоника` – има съвпадение "5те пици," ("3те тоник" не е съвпадение, защото "к" не влиза в класа **`\W`**)
`три часа и половина` – няма съвпадение (няма цифра отпред)

Шаблон: **`[\D].[\w]`** (както и **`[^\d].[\w]`**)
`5та` – няма съвпадение (първият символ трябва да не е цифра)
`т.5` – има съвпадение "т.5" (но точката си остава специален символ!)

Шаблон: **`[^\d\s]`**
`a` – има съвпадение "a" (един символ, който не е цифра и whitespace)
`4` – няма съвпадение

Шаблон: **`[\D\S]`**
`a` – има съвпадение "a"
`4` – има съвпадение "4" (търсим символ, който не е цифра, **или** символ, който не е празно пространство – "4" не е празно пространство)

Както се вижда, **[\D]** е същото като **[^\d]**, но **[\D\S]** не е същото като **[^\d\s]**. В подобни случаи трябва да внимаваме какво точно имаме предвид в шаблона.

Метасимволи за количество

Едни от най-често използваните метасимволи са тези за количество повторения на даден подниз или символ в шаблона (**quantifier metacharacters**). С няколко такива метасимвола вече се сблъскахме в горните примери, а сега ще ги разгледаме по-подробно.

Символът * – нула или повече повторения

С ***** означаваме 0 или повече повторения на символа (или метасимвол, включително клас от символи), предхождащ знака *****:

Шаблон: **бира***
разбирам – има съвпадение "бира"
бираааааа! – има съвпадение "бираааааа"
бирррррра! – има съвпадение "бир" (***** важи само за символа "а")
бирено коремче – има съвпадение "бир" (0 повторения на "а")

Ако преди звездичката има клас от символи, то търсим поредица от последователни срещания на символи от този клас. Не се изисква обаче това да са повторения един и същи символ от този клас. Ще напомним и че звездичката вътре в квадратните скоби губи специалното си значение:

Шаблон: **ба[ла]*йка**
балалайка – има съвпадение "балалайка"
баллалаалайка – има съвпадение "баллалаалайка"
баалайка – има съвпадение "баалайка"

Шаблон: **ба[л*а*]йка**
балалайка – няма съвпадение (трябва "л", "а" или "*" по средата)
байка – няма съвпадение
ба*йка – има съвпадение "байка"

Горният пример показва, че не можем да използваме квадратните скоби и звездичката, за да означим повторение на някой от символите в класа на квадратните скоби. Ако искаме няколко пъти "**а**" или няколко пъти "**б**", не ни върши работа нито "**[а*б*]**" (търси само веднъж "**а**", "**б**" или "*****"), нито "**[аб]***" (което пък намира например "**аббаб**"). В такива случаи ще ни трябва специалният символ за алтернативен избор, който също ще разгледаме малко по-късно. Ако пък искаме да се повтаря точно определена редица символи, можем да използваме метасимвола за група и да оградим редицата в скоби: **(редица)***. Това групиране има и други приложения, за които подробно ще говорим по-късно, но засега е важно да знаем, че то

ни позволява да отделяме логически части от шаблона и да прилагаме някой специален метасимвол върху цели такива части:

Шаблон: **ба(ла)*йка**
байка – има съвпадение "байка"
балалайка – има съвпадение "балалайка"
баллалайка – няма съвпадение ("лла" го няма в шаблона)

Символът + – едно или повече повторения

Този метасимвол е идентичен със символа *****, като единствената разлика между двата, е, че **+** изисква задължително поне едно повторение на конструкцията, за която се отнася – т.е. той съвпада 1 или повече повторения.

Шаблон: **бира+**
бираааааа! – има съвпадение "бираааааа"
бирено коремче – няма съвпадение (трябва поне едно "а" след "бир")

Символът ? – нула или едно повторения

Въпросителният знак е метасимвол, който означава 0 или 1 повторения. Обикновено се използва за някаква незадължителна конструкция в шаблона, която или се среща само веднъж, или изобщо не се среща. В примера отново използваме ограждането със скоби за цяла група.

Шаблон: **няма(бира)?**
няма бира – има съвпадение "няма бира"
бира няма – има съвпадение "няма"

Метасимволи за точен брой повторения

Ако искаме да укажем с по-голяма точност броя на последователните срещания на даден символ или конструкция, можем да използваме специалните символи за точен брой повторения. С **{n}** указваме, че прехождащият (мета)символ ще се повтаря точно **n** пъти. **{n,}** означава поне **n** повторения, а **{n,m}** е за поне **n**, но не повече от **m** последователни срещания на дадената конструкция:

Шаблон: **бир{2,3}а***
разбирам – няма съвпадение ("р" трябва да се среща поне 2 пъти)
бирраааааа! – има съвпадение "бирраааааа"
бирррррра! – има съвпадение "биррр" ("р" се среща 4 пъти, конструкцията позволява до 3 и взима максималния брой)

Лесно можем да забележим, че познатите ни `*`, `+` и `?` могат да се представят чрез символите за точен брой повторения, както следва: `*` като `{0,}`, `+` като `{1,}` и `?` като `{0,1}`.

"Мързеливи" метасимволи за количество

По-рано казахме, че регулярният израз винаги връща най-лявото съвпадение. С въвеждането на метасимволите за количество обаче виждаме, че има възможност две съвпадения да започват от една и съща позиция и просто да са различно дълги. Кое се избира тогава?

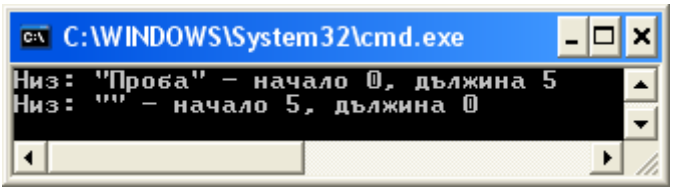
Отговорът е, че метасимволите за количество са "лакоми". Това означава, че когато търсим съвпадения с шаблон, съдържащ тези метасимволи, машината за регулярните изрази се опитва да намери максималния (като дължина) подниз, който удовлетворява условието. Ако от една и съща позиция започват два различни подниза, отговарящи на шаблона, като съвпадение при търсене се получава винаги по-дългият. При последващо търсене, както знаем, се продължава от позицията, на която е завършило предишното търсене. За по-голяма яснота ще разгледаме следния пример.

Пример за "лакомо" съвпадение

Регулярен израз – "**`\w*`**".

Низ, в който ще търсим – "**`Проба`**".

Ето какво ще получим в резултат:



Как можем да разтълкуваме полученото? Да си припомним какво казахме за начина на откриване на съвпадения по-рано в секцията за литералите. Машината за регулярните изрази започва търсенето си от позиция 0. Най-дългият низ, който може да се получи в съответствие с шаблона, е целият низ "**`Проба`**", което и виждаме като първи резултат.

Понеже търсенето продължава оттам, откъдето завършва предното съвпадение, то остава да се търси само в позиция 5 – където вече свършва низът. Сега началото и краят на всяко възможно съвпадение ще са на една и съща позиция, и следователно тук машината може да намери съвпадения единствено с празния низ. Празният низ обаче също отговаря на шаблона, ето защо се открива като съвпадение и го виждаме в резултата.

Ако бяхме използвали шаблона "**`\w+`**", то празният низ нямаше да бъде разпознат, защото не съдържа нито един **alphanumeric** символ, и "**`Проба`**" щеше да е единственият намерен подниз.

Това, което трябва да запомним, е че търсене не се провежда от всяка възможна позиция, а се избира едно съвпадение (при "лакомите" метасимволи за количество това е най-дългото възможно съвпадение) и търсенето продължава от неговия край. Ето защо понякога може да се случи да изгубим резултат, който ни интересува.

Нежелано "лакомо" съвпадение

Как можем да променим този механизъм на търсене, ако искаме да намерим низ, който не е максималното съвпадение? Единият начин е да използваме метасимволите за точен брой повторения. Този подход обаче не винаги е в състояние да реши проблема.

Да разгледаме следния пример – искаме да извадим HTML тагове от някакъв HTML документ. По-конкретно, нека имаме низа "** This is a paragraph tag **". Нашата задача е да открием таговете ** ** и ** **. Лесно се вижда, че това не може да стане с израза **<.+>**, за който може би бихме се сетили първо. Понеже метасимволът **+** е "лаком", търсенето по този израз ще намери първо като съвпадение направо целия низ и ние няма да извлечем желания резултат.

Не можем да приложим и символите за точен брой повторения – за долна граница можем да сложим 1, но не знаем каква горна граница да зададем. Тук тагът е еднобуквен, но в HTML може да има много разнообразни по дължина тагове. Дори да зададем някаква надвишаваща и най-дългия таг горна граница, това не ни води до вярното съвпадение, защото в общия случай пак можем да съвпаднем и текста между отварящия и затварящия таг, а освен това символът **>** може и отделно да присъства в този текст.

"Мързеливо" съвпадение

Решението на проблема тук е в конструкцията за т. нар. "мързеливо" съвпадение. С добавяне на символа **?** след всеки от метасимволите за количество, ние принуждаваме машината на регулярните изрази да приеме първото възможно (съответно и най-кратко) съвпадение. Да използваме тази конструкция в нашия пример – да трансформираме израза в **<.+?>**. Сега вече получаваме точно търсения резултат - "** **" и "** **".

"Мързеливото" съвпадане може да се прилага за всички споменати специални символи за количество – ***?**, **+?**, **??**, **{n}?** и т.н. Разгледани отделно, изглежда, че тези конструкции просто могат да се заместят с долните граници за брой повторения на съответните метасимволи (0 за ***** и **?**, 1 за **+** и т.н.), но както видяхме, това не е точно така. Ситуацията се променя, когато мързеливите метасимволи са последвани от друга част от шаблона, която трябва да съвпадне. Тогава минималният брой повторения са повторенията в низа до достигането на тази друга част от шаблона.

Метасимволи за местоположение

Метасимволите за местоположение (**zero-width assertions** или **anchors**) се различават от вече разгледаните типове, защото не се използват за съвпадане на символи в текста, а за съвпадане с позиция в текста. Те обикновено се използват, за да укажат, че дадена поредица от символи трябва да се намира на някакво точно определено място в текста. Съвпаденията, които намират, са с нулева дължина и не карат машината на регулярните изрази да премине на следващия символ. Какво означава това на практика, ще разберем след като разгледаме примерите за отделните метасимволи.

Символът ^

С помощта на този символ можем да поискаме съвпадението да се намира в началото на низа. Символът **^** е шаблон за позицията преди първия символ.

Шаблон: **^бира**
разбирам – няма съвпадение (поднизът "бира" не е в началото)
бираааааа! – има съвпадение "бира"
бирено коремче – няма съвпадение (няма "бира" в началото)

Символът $

Аналогично, символът **$** изисква съвпадение с края на низа (позицията след последния символ). Има едно изключение от това правило и то е ако низът завършва със символа за нов ред **\n**. Тогава **$** намира съвпадение и с позицията преди този символ:

Шаблон: **бира$**
разбирам – няма съвпадение (поднизът "бира" не е в края)
хубава бира – има съвпадение "бира"
скарата гълта много бира\n – пак има съвпадение "бира" (има само "\n" до края)

Символите ^ и $ в многоредов режим

Често когато четем например текст от файл, където има много нови редове, се интересуваме от позицията на търсения низ в рамките на реда, а не на целия текст. Синтаксисът на регулярните изрази позволява символите **^** и **$** да означават съответно началната и крайната позиция не само на целия низ, в който търсим, а и на даден ред от него. Тогава:

Шаблон: **^бира**
лято е.\n
бирата е студена – има съвпадение "бира"
лято е,\n

но бирата е студена – няма съвпадение ("бира" не е в начало на ред)
Шаблон: **\w{4}\?$**
какво лято?\n
каква бира? – две съвпадения "лято" и "бира" (4 alphanumeric символа и въпросителна на края на ред)

За да използваме тази функционалност на символите **^** и **$**, трябва първо да активираме опцията **Multiline** при търсенето. Как става това, ще разберем в частта за опциите, по-нататък в темата. Да обърнем внимание, че тази опция не е включена по подразбиране.

Употреба на символите за край и начало

Основното приложение на тези метасимволи е при валидация на потребителски вход, както ще се убедим не след дълго. Поставяйки нашия шаблон между символите за начална и крайна позиция, ние на практика задължаваме целия текст да отговаря на шаблона. Ако не го направим, регулярният израз търси съвпадение в произволна част от текста и може да се окаже така, че само тази част отговаря на валидиращите условия, което не е коректно решение на задачата за валидация.

Разбира се, метасимволите **^** и **$** могат да се употребяват и за много други цели. Може да ни се налага например да търсим първите думи на редовете в дълъг текст и да променяме първата им буква на главна.

Един друг пример използва важна особеност на тези символи – те могат да намират като съвпадение празния низ (това е вярно още единствено за конструкциите **.*** и **.?**). Това понякога не е удобно и трябва да внимаваме, но ако искаме например да добавим нещо в началото на всеки ред (подобно на символите ">", които някои уеб-базирани системи за изпращане на поща слагат автоматично при цитиране на писмото в отговора), можем да се възползваме именно от съвпадането на празния низ (ще използваме метода **Replace(…)**, за който ще стане дума по-късно):

```
Regex.Replace(text, @"^", ">", RegexOptions.Multiline);
```

Някои особености

Добре е да се запомни, че символите **$** и **^** трябва да бъдат escape-вани навсякъде в регулярния израз, ако искаме да ги представим като литерали. На пръв поглед изглежда, че те имат смисъл само в началото и в края на шаблона и ако не са на тези позиции, няма нужда да се третират като специални, но това не е така заради функцията **Multiline**.

Една последна важна бележка се отнася до символа за край на ред/текст **$**. Трябва да се внимава с позицията, на която той намира съвпадение, защото тя не е валидна позиция в низа. Така че ако се опитаме да използваме позицията на съвпадението като индекс върху низа, ще се сблъс-

каме с изключение, защото сме излезли извън границите му. Следният пример ще доведе до грешка:

```
string text = "тестов низ";
string pattern = "$";
Match match = Regex.Match(text, pattern);
char matchIndex = text[match.Index];
```

Още не сме обяснили класовете за работа регулярни изрази в .NET, но се вижда, че се опитваме да осъществим достъп до несъществуващ елемент на низа. Подобно нещо може да се получи и с изразите "^" и "**^$**", ако сме в многоредов режим и низът завършва със символа за нов ред (да си спомним, че тогава символът ^ намира съвпадение след всяко **\n**).

Символите \A, \Z и \z

Символите **\A** и **\Z** напълно отговарят по функционалност съответно на символите **^** и **$**, с тази разлика, че не позволяват многоредовото съвпадане. Независимо дали е активирана опцията **Multiline**, те винаги откриват за съвпадение единствено началото и края на целия низ, а не на всеки ред.

Символът **\z** се различава от **\Z** и от **$** по това, че той не открива съвпадение преди "**\n**" на края на низа:

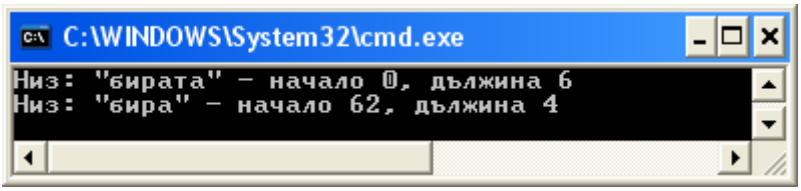
Шаблон: **бира\Z**
скарата гълта много бира\n – има съвпадение "бира"

Шаблон: **бира\z**
скарата гълта много бира\n – няма съвпадение

Символът \b – граници на думи

Специалният символ **\b** се използва за указване на позиция "на границата на дума". По-точно той е шаблон за съвпадение с нулева дължина, който намира съвпадения в позициите между символ от класа **\w** (**alphanumeric** символи, които най-общо съставят думите) и символ от класа **\W**. Съвпадение с **\b** има и в началото или края на низа, ако той започва, респективно завършва с **alphanumeric** символ.

Малко примери – нека търсим с израза "**\bбир\w***" в низа "**бирата вече не е топла, набираме доброволци да търсят студена бира**". Изходът тук е следният:



Какво става при търсенето в този пример? Машината започва да търси съвпадения с първия символ от регулярния израз, който е **\b**. Тъй като в низа първият символ е alphanumeric, то началото му се счита за успешно съвпадение. Нататък машината трябва да провери за подниз "**бир**" и остатък от прилежащата дума (нула или повече word characters). Така първата дума – "**бирата**" – се хваща от шаблона. Това става и при последната дума, където метасимволът **\b** съвпада с позицията след интервала и преди "**б**" (защото единият е от клас **\w**, а другият – от **\W**). Думата "**набираме**" обаче не е валиден екземпляр на шаблона, защото позицията преди подниза "**бир**" не се намира на границата на дума.

Символът \B

Символът **\B** намира съвпадение навсякъде, където **\b** не намира, тоест на позиции между два символа от клас **\w** или между два символа от клас **\W**. Началото и краят на низ, започващ (респективно свършващ) със символ, който не е alphanumeric/word character, също са съвпадения за този метасимвол.

Метасимволът за избор |

Конструкция от вида "**regexp1|regexp2**" означава, че успешно съвпадение ще бъде открито както ако низът отговаря на шаблона "**regexp1**", така и ако отговаря на шаблона "**regexp2**". На практика с метасимвола **|** даваме на машината за регулярни изрази възможност за избор. Ако намери съвпадение с израза от лявата страна, шаблонът е удовлетворен. Ако това не стане, машината опитва да намери съвпадение с израза от дясната страна, с което може и да успее.

Трябва да отбележим, че символът **|** има най-нисък приоритет при регулярните изрази и търси съвпадение с **целия** подниз-шаблон от лявата си страна и **целия** подниз от дясната. Ако искаме да променим това поведение, трябва да използваме групиране (което ще обясним по-късно), за да ограничим видимостта на метасимвола. При групиране, символът **|** действа в границите на групата.

Шаблон: **бира|биричка**
хубава бира – има съвпадение "бира"
биричката става – има съвпадение "биричка"
бирено коремче – няма съвпадение
бирбиричка – няма съвпадение (**|** не е само между "а" и "б")

Шаблон: **вишн(а|я)та**
вишната – има съвпадение "вишната"
ята – няма съвпадение (**|** е само между "а" и "я")

Повече възможности

Разбира се, метасимволът за алтернативен избор може да се използва последователно нееднократно, за да означим повече от две възможности.

Шаблон: **вишн(а|я|и|овка)**
вишничка – има съвпадение "вишни"
вишновка – има съвпадение "вишновка"

Шаблон: **\bвишн(а|я|и|овка)\b**
вишничка – няма съвпадение (след второто "и" трябва граница на дума)
вишновка – има съвпадение "вишновка"

Както се вижда от първия пример, трябва да внимаваме да не би да търсим погрешно за части от думи ("**вишни**" във "**вишничка**"), а всъщност да искаме цели думи (само "**вишни**"). Решението се вижда във втория пример – добавяме символа за граница на дума.

Можем да забележим, че с единични символи конструкцията "**(а|б|в|г)**" не е по-различна от "**[абвг]**", но силата на алтернативния избор е, че можем да описваме много по-разнообразни възможности от единичен символ или клас от символи. Можем също да влагаме вътрешни шаблони като избори (чрез групиране, за което ще разкажем по-късно) и т.н.

По-обстоен пример с разгледаните метасимволи

Нека сега се опитаме да обединим казаното дотук в един по-голям регулярен израз и да разгледаме прилагането му върху някой низ:

Шаблон: **^.+?\s\w*((.e)+н)?\s+\w*\s.+?(\d){2,5}.*?[БИРА]+**
Низ: **това изречение има от две до пет цифри тук: 346; и може би РИБА, БИРА или БАР..**

Да започнем да прилагаме шаблона. Веднага виждаме, че съвпадението с шаблона трябва да започва от началото на низа заради символа **^**. По-нататък машината започва да прилага възможните варианти на шаблона:

Шаблон: **.+?**
Съвпада с: **"това"**
Коментар: Използваме мързеливото съвпадане с точка. Ако не направим това, символът **+** става "лаком" и ще поеме сам целия низ, защото точката отговаря на всеки символ от него

Шаблон: **\s**
Съвпада с: **" "**

Шаблон: **\w***

Съвпада с: **"изречение"**
Коментар: Тук не сме използвали символа **?** за мързеливо съвпадение и в резултат символът звезда става "лаком". Може да се заблудим, че съвпадението с него ще е само "из", защото следващата част от шаблона отговаря на подниза "речен", но това не е така. Лакомата звездичка ще приеме за съвпадение всичко до следващото празно място (което вече не се вписва в шаблона **\w***), т.е. цялата дума "изречение"

Шаблон: **((.е)+н)?**
Съвпада с: **""**
Коментар: Тази част от шаблона трябва да търси за повторения на конструкцията символ+"е", последвани от "н", но в момента машината на регулярните изрази се намира на празното място след думата "изречение". Въпреки че вече сме подминали подниз, който отговаря на частта от шаблона, оттук надясно не следва друг такъв подниз. Понеже сме използвали символа за незадължително срещане **?**, съвпадението остава единствено празният низ. Ако не бяхме сложили **?**, резултатът би бил, че няма никакво съвпадение с целия шаблон, защото машината ще се премести да търси отначало от втория символ в низа, но там вече не е началото и съвпадането с **^** винаги ще пропада.

Шаблон: **\s+\w*\s**
Съвпада с: **" има "**
Коментар: Тук например вече не е от значение дали използваме "лакомо" или "мързеливо" търсене. Има само един единствен вариант, който може да бъде приет за съвпадение и това е точно посоченият. Обърнете отново внимание, че "мързеливото" съвпадане е важно тогава, когато след съответния количествен метасимвол имаме указана конкретна част от шаблона, за която да проверим (както беше преди малко). Тогава "лакомото" търсене поглъща и тази част и нашата проверка не е валидна.

Шаблон: **.+?\d{2,5}**
Съвпада с: **"от две до пет цифри тук: 346"**
Коментар: Ето ново потвърждение на горния коментар. Конструкциите **.+** и **.*** винаги поглъщат всичко до края на низа. Когато обаче ги ограничим с **?**, ще получим най-малкото възможно съвпадение, така че следващата част от шаблона да продължи да отговаря на низа, както е в случая – искаме от две до пет десетични цифри.

Шаблон: **.*?[БИРА]+**
Съвпада с: **"; и може би РИБА"**
Коментар: **.*?** отговаря на всички символи до главната буква "Р", която е валидно съвпадение с частта **[БИРА]+**. Тук обаче имаме "лаком" плюс и машината на регулярните изрази продължава да търси напред по низа за срещания на някой от изброените в квадратните скоби символи. Това

продължава до запетаята, която вече не е такъв символ. На това място целият шаблон е открил едно пълно съвпадение. Краят на низа не е достигнат, но тъй като не сме сложили знак **$** или **\z**, това не е пречка. Ако сега се опитаме да търсим повторно (от вече достигнатото място – позицията на запетаята), разбира се няма да получим ново съвпадение, тъй като нашият шаблон започва с **^**, а ние сме минали началото.

Регулярните изрази в .NET Framework

И така, след като разгледахме основните моменти от синтаксиса на регулярните изрази, е време да видим как можем да използваме техния апарат със средствата на .NET Framework. Класовете за работа с регулярни изрази се намират в пространството от имена **System.Text.RegularExpressions**. Както ще се убедим, с помощта на методите на тези класове работата с шаблони никак не е сложна. Обикновено от нас се изисква само да инициализираме един обект с даден регулярен израз. След това с последователни извиквания в цикъл на метода за търсене на съвпадение лесно получаваме всички необходими резултати.

Разбира се, .NET Framework предлага и достатъчно възможности за по-задълбочено боравене с регулярните изрази, например за работа с групи, за заместване по даден шаблон, за разделяне на низ по шаблон и т.н. Има и възможности за настройка на различни опции при търсенето на съвпадения. Ще направим кратък преглед на класовете за работа с регулярни изрази, а след това ще разгледаме по-подробно методите и свойствата им, както и тяхното приложение.

Пространството System.Text.RegularExpressions

Класът **Regex** е основният клас за работа с регулярни изрази. Той представя един неизменим (константен) шаблон. Чрез неговите методи се извършват операциите с регулярни изрази – търсене, заместване, разделяне на низ по регулярен израз и др. Класът **Regex** може да се използва както чрез свой обект, инициализиран с даден шаблон, така и чрез статичните си методи.

Класът **Match** представя едно съвпадение при търсене с регулярен израз. Чрез свойствата си той ни дава цялата необходима информация за съвпадението – текста му, дължината му и началната му позиция в низа и. Методите на класа **Match** позволяват да се преминава към следващото намерено съвпадение.

Класът **MatchCollection** съдържа списък от всички съвпадения, получени при търсенето с шаблона върху израза. Чрез методите за търсене на класа **Regex** можем да получим като резултат именно такъв обект, който после да обходим и да обработим всяко съвпадение.

Класовете **Group** и **Capture**, както и съответните класове за колекции **GroupCollection** и **CaptureCollection**, са свързани с механизма на групите в регулярните изрази, с който ще се запознаем след малко. Техните методи ни дават редица възможности за работа с тези групи.

Делегатът **MatchEvaluator** се използва при операциите за заместване, като имаме възможност да прилагаме дефинираната в него потребителска функционалност върху всяко намерено съвпадение и да заместваме с получения резултат.

Изброеният тип **RegexOptions** съдържа различни константи, които се използват за указване на опции при търсенето с регулярни изрази.

В пространството **System.Text.RegularExpressions** се намира и класът **RegexCompilationInfo**. Той се използва в процеса на компилиране на регулярните изрази в самостоятелни асемблита, за което ще стане дума по-нататък в темата. Няма да го разглеждаме подробно.

Представяне на шаблони

Както вече споменахме, за представяне на регулярен израз в .NET служи класът **Regex**. Ето защо той е задължителен при работа с регулярни изрази. Можем да го използваме по два начина.

Единият вариант е да инстанцираме обект от този клас и да подадем на конструктора му като параметри низ за шаблон и евентуално някакви опции. След това можем да извикваме методите на обекта, които реализират операциите с регулярни изрази.

Другият вариант е да използваме статичните методи, които **Regex** предлага, като всеки път им подаваме необходимия шаблон за параметър.

Тези две възможности ще демонстрираме със следния кратък пример:

```
string text = "0887-654-364";
string patter = @"088\d-\d{3}-\d{3}";

// Instance Regex
Regex regex = new Regex(pattern);
Match match = regex.Match(text);

// Static Regex
Match match2 = Regex.Match(text, pattern);
```

Двата начина за работа с класа са приблизително еквивалентни и е въпрос на личен избор кой да се използва. Статичните методи предлагат пълноценен достъп до основната функционалност за работа с регулярни изрази, тъй като повечето методи имат статичен аналог. В повечето случаи можем спокойно да използваме тях, защото така не създаваме излишни обекти. Ако обаче се нуждаем от по-специалните вариации на методите, е по-добре да създадем инстанция на **Regex**. Вариантът с обект

е по-добрият избор и ако се налага да използваме един и същи шаблон няколкократно върху различни низове, тъй като така шаблонът се компилира за машината на регулярните изрази само веднъж и това подобрява ефективността.

Ако изберем работата с обект и по някаква причина искаме да извлечем регулярния израз, с който е инстанциран този обект, можем да използваме метода **`ToString()`**. Той, както и простото преобразуване към **`string`**, ни връщат точно желания резултат.

Търсене с регулярни изрази

Основната процедура, свързана с регулярните изрази – търсенето на съвпадения с даден шаблон в даден низ – извършваме с помощта на класовете **`Regex`** и **`Match`** и техните методи и свойства. Търсенето за съвпадения можем също да извършим по два начина. Единият е да търсим последователно няколко пъти за всяко следващо успешно съвпадение. Това можем да постигнем чрез метода **`Match(…)`** на класа **`Regex`**. Методът **`Matches(…)`** пък ни дава възможност да извършим всички търсения до изчерпването на низа наведнъж и да получим съвпаденията в колекция от тип **`MatchCollection`**.

Двата метода на пръв поглед действат различно, но машината за регулярните изрази се държи по познатия начин и при двата – всяко следващо търсене започва от края на предното. Разликата е, че в единият случай трябва да предизвикваме всяко търсене (докато стигнем до неуспешно съвпадение, т.е. низът е свършил) с извикване на метод, а в другия това става автоматично. Понеже често не се нуждаем от всички съвпадения, и двата метода намират своето приложение.

Няколко основни правила при търсенето

Хубаво е да запомним следните няколко практически съвета, които обикновено са свързани с търсенето по шаблон:

- Търсенето с регулярни изрази в повечето случаи е свързано с извличане на информация от текст. Хубаво е да подберем шаблона така, че след като открием съвпадение, лесно да можем да го обработим, за да получим информацията, която практически ни върши работа. Оптималният вариант, разбира се, е самото съвпадение да е това, което ни трябва, но не винаги ситуацията е толкова проста.
- Когато не извличаме информация, ние обикновено проверяваме дали изобщо съществува съвпадение. Този подход се прилага с цел валидация на данни или при класифициране на различни текстове въз основа на това какви низове съдържат, и др.
- В повечето случаи не се налага да допълваме шаблона със символи, които трябва да "поемат" частта от текста, която не ни интересува. Не е нужно шаблонът ни да покрива целия текст. Понеже обикно-

вено просто търсим някакви поднизове, то естествено е той да отразява само тях.

- Трябва да се обърне специално внимание на "лакомите" символи при съставянето на шаблона. Те често могат да ни лишат по невнимание от желаната информация. Подобен ефект може да се получи, ако не оценим правилно принципа на действие на машината на регулярните изрази. С придобиването на опит ще правим все по-малко грешки от такъв тип.
- В някои практически проблеми се налага работа с по-необичайни символи (букви с ударения, нестандартни знаци и т.н.). Има специални начини за действие с Unicode, които ще разгледаме към края на темата.

Класът Match

Класът **Match**, както вече споменахме, представя всяко съвпадение на шаблон с подниз. Информацията за съвпадението е достъпна чрез свойствата на **Match**:

- Свойството **Success** връща булева стойност, която показва дали съвпадението е успешно. Неуспешното съвпадение също е обект от класа **Match**, който може да бъде върнат от метода **Match(…)** на класа **Regex**, когато при търсене се стигне до края на низа и не е намерено нито едно съвпадение с шаблона.
- Свойството **Value** съдържа стойността на съвпадението, тоест подниз, който отговаря на нашия шаблон. Същата стойност се връща и от метода **ToString()**, както и от преобразуването към **string**.
- Свойството **Index** представя позицията в низа, от която започва съвпадението (броенето е от 0).
- Свойството **Length** ни дава дължината на съвпадението.

Класът **Match** няма **public** конструктор и съвпадението, което пази, не може да се изменя. Обекти от този тип можем да получим единствено чрез методите на класа **Regex** и те пазят вътрешна информация за регулярния израз, при търсенето с който са получени.

Статичното свойство **Match.Empty** представлява обект – неуспешно съвпадение. Той може да се използва за сравнение (дали нашето съвпадение е успешно), но по-добре е да се използва свойството **Success**. Свойството **Empty** има нулеви стойности за **Value**, **Index** и **Length**.

Типът **MatchCollection** ни предоставя колекция от обекти на класа **Match**. Такива колекции можем да получим като резултат от изпълнение на метода **Matches(…)**, когато търсим всички съвпадения наведнъж. Колекцията се итерира по стандартните начини (например с **foreach**), а свойството **Count** ни дава броя успешни съвпадения.

Последователно еднократно търсене с Match(...) и NextMatch()

Методът **Match(text)** на класа **Regex** е основният метод, който се използва при работата с регулярни изрази. Той извършва едно търсене за съвпадения с шаблона в низа. Действието му се прекратява при първото намерено съвпадение и се връща обект от класа **Match**, който описва съвпадението. Такъв обект се връща и ако не е намерено нито едно съвпадение, така че за да проверим дали все пак имаме валиден резултат, трябва да използваме свойството **Success** на класа **Match**.

Статичният вариант на този метод е **Match(text, pattern)**. Други варианти (но не статични) позволяват да се указва определена част от низа, в която да се търси за съвпадение.

Чрез обекта от клас **Match**, който сме получили като резултат, можем да продължим търсенето до намирането на ново съвпадение. Това става чрез метода **NextMatch()**. Той връща нов **Match** обект, който представя следващото съвпадение в низа. Това е възможно, тъй като в класа **Match** се пази информация за шаблона, с който търсим, както и за това къде започва и колко е дълго текущото съвпадение. Търсенето продължава от позицията, на която то завършва. Ако извикаме **NextMatch()** за обект, който е неуспешно съвпадение, ще получим просто още едно неуспешно съвпадение.

Обикновено използваме един обект от тип **Match** (резултата от метода **Match(…)**), на който в цикъл присвояваме последователно резултата от поредното извикване на **NextMatch()**, след което го обработваме. За условие на цикъла обикновено използваме проверка на свойството **Success**, но в тялото може да правим и други проверки, въз основа на които евентуално да прекратим цикъла.

Тагове за хипервръзки в HTML код – пример

Казаното дотук ще илюстрираме с един практически пример. В редица ситуации се налага да се извлекат хипервръзките от даден HTML документ. Например по подобен начин действат т.нар. web-spiders, които се използват от Интернет търсачките за обхождане на голямо количество страници за кратко време.

Можем да използваме следния регулярен израз за извличане на хипервръзките:

```
<\s*a\s[^>]*?\bhref\s*=\s*('[^']*'|""[^""]*""|\S*)[^>]*>(.|\s)*?
<\s*/a\s*
```

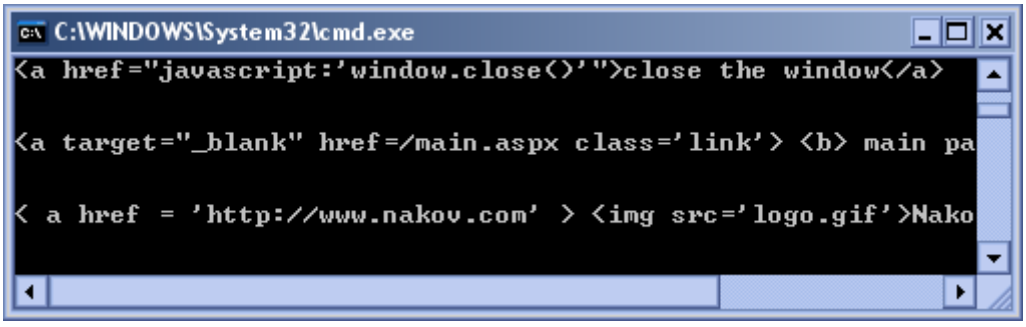
Чрез следните стъпки ще създадем програма, която да открива таговете от вида "**…**", които представляват хипервръзки:

1. Отваряме VS.NET и създаваме нов конзолен проект.

2. Въвеждаме кода на програмата:

```
static void Main()
{
   string text = @"<html>
      This is a hyperlink:
      <a href=""javascript:'window.close()'"">
      close the window</a><br> ... and one more link: <a
      target=""_blank"" href=/main.aspx class='link'> <b>
      main page</b> </a>< a href = 'http://www.nakov.com'
      > <img src='logo.gif'>Nakov's home site < /a >";

   string hrefPattern = @"<\s*a\s[^>]*?\bhref\s*=\s*" +
      @"('[^']*'|""[^""]*""|\S*)[^>]*>" +
      @"(.|\s)*?<\s*/a\s*>";
   Match match = Regex.Match(text, hrefPattern);
   while (match.Success)
   {
      Console.WriteLine("{0}\n\n", match);
      match = match.NextMatch();
   }
}
```

3. Сега можем да стартираме програмата. Резултатът от нея е следният:



Как работи нашата програма? Търсенето с регулярния израз е по стандартната схема с метода **`NextMatch()`** в цикъл, която вече обяснихме. Стойностите на съвпаденията извеждаме чрез преобразуване до **`string`**. Бихме могли да използваме и свойството **`Value`**. Ще обясним действието на самия шаблон, подобно на вече разгледания по-горе пример:

Шаблон: **`<\s*`**
Коментар: Започва със символа "**<**" и преминава през празното пространство след него (ако има).

Шаблон: **`a\s`**
Коментар: Търси символ "**a**", следван задължително от празно пространство.

Шаблон: **[^>]*?\bhref**
Коментар: Преминава през неопределен брой символи (но различни от затварящия таг "**>**", докато намери дума "**href**" (ако тагът има други атрибути преди "**href**", ги пропуска). Метасимволът ***** е "мързелив", защото конструкцията е последвана от подниз, който не искаме да изпуснем (въпреки това програмата ще работи и с "лаком" плюс – прегледайте целия шаблон внимателно и помислете защо!).

Шаблон: **\s*=\s***
Коментар: Търси символа "=", евентуално предшестван и следван от празно пространство.

Шаблон: **'[^']*'|""[^""]*""**
Коментар: Ако следват двойни кавички или апостроф, преминава през 0 или повече символа до намиране на съответни затварящи двойни кавички или апостроф.

Шаблон: **|\S***
Коментар: Ако не следват двойни кавички или апостроф, преминава през 0 или повече символа, различни от празно пространство. Цялата конструкция за алтернативен избор е затворена в групиращи скоби, за да ограничи видимостта на метасимвола **|**.

Шаблон: **[^>]*>**
Коментар: Пропуска всички символи до намиране на символ "**>**" и преминава през него (включително други атрибути на тага). Тук вече ***** спокойно може да е лаком, защото е следван от символ, който самият той не може да "поеме" (защото е забранен в класа от символи, към който се отнася звездичката).

Шаблон: **(.|\s)*?**
Коментар: Преминава през 0 или повече произволни символи, които представляват текста на хипервръзката. Звездичката отново е "мързелива", защото иначе ще приеме целия низ до края за съвпадение. Конструкцията **(.|\s)** отговаря на абсолютно всеки възможен символ (точката е за всички без нов ред, който пък е съвпадение за **\s**).

Шаблон: **<\s*/a\s*>**
Коментар: Търси затварящ таг "****", евентуално съдържащ на места разделящи символи празно пространство (whitespace).

Още нещо за позицията на следващото търсене

Има един случай, когато правилото "новото започва от края на старото" не е в сила. Когато съвпадението е било празен низ (т. е. свойството

Length съдържа 0), машината на регулярните изрази премества текущата позиция с едно напред. Ще покажем това с един кратък пример:

```
string text = @"Testing this and that";
Match match = Regex.Match(text, @".*?");
if (match.Success)
{
   Console.WriteLine("Съвпадение: <>{0}<> на позиция {1}",
      match.Value,match.Index);
   match = match.NextMatch();
   if (match.Success)
   {
      Console.WriteLine("Съвпадение: <>{0}<> на позиция {1}",
         match.Value, match.Index);
   }
}
// Output:
// Съвпадение: <><> на позиция 0
// Съвпадение: <><> на позиция 1
```

Тук "мързеливото" **.*?** открива най-късото съвпадение, което е винаги празният низ, който се среща на всяка позиция. Макар че първото съвпадение от позиция 0 има дължина 0, следващото търсене започва вече от позиция 1, защото в противен случай машината просто би търсила до безкрай. Така ако извикаме достатъчен брой пъти **NextMatch()**, ще се стигне до края на низа и търсенето ще завърши при проверката за успешно съвпадение, която ще се провали.

Търсене за съвпадения наведнъж с Matches(...) и MatchCollection

Другият вариант за търсене, който най-често се използва по подразбиране в някои езици за програмиране, например Perl, е да използваме метода **Matches(…)**. Той извършва цялостно търсене по шаблона, т.е. машината на регулярните изрази спира последователните търсения едва при достигане на края на низа. Както казахме, получените съвпадения при търсенето се запазват под формата на обект от класа **MatchCollection**. Тези съвпадения не могат да са част едно от друго поради факта, че всяко следващо започва от края на предното. Ще разгледаме един кратък пример, който демонстрира и този начин на работа. Колекцията обикновено обхождаме с цикъл **foreach**, вътре в който ще изведем съвпаденията:

```
static void Main()
{
   // A pattern for cyrillic words
   Regex regex = new Regex(@"\b[А-Яа-я]+\b");

   String text =
```

```
      "The Bulgarian word 'бира' (beer) often" +
      " comes with the word 'скара' (grill).";

    MatchCollection matches = regex.Matches(text);
    foreach (Match match in matches)
    {
      Console.Write("{0}:{1} ", match);
    }

    // Output: бира скара
}
```

Методът **`Matches(…)`** се използва, когато се нуждаем от всички съвпадения. Ако имаме причина да прекратим търсенето, преди да сме обходили целия низ, по-добре е да приложим схемата с **`Match(…)`** и **`NextMatch()`**, за да не губим производителност.

Групи в регулярните изрази

На няколко места дотук в темата споменавахме за групиране и групи в регулярните изрази. Всъщност групите са една широко използвана конструкция, която позволява логическо обособяване на части от шаблона, както и извличане на различни видове информация именно като отделни части от общия регулярен изрази.

Създаването на група в шаблона вече описахме няколкократно. Достатъчно е просто да оградим част от израза в кръгли скоби – "**`Това е израз с (група)`**".

Предимства на групите

Предимствата на групите са няколко. Те позволяват някои метасимволи (за количество, за избор и др.) да се прилагат върху цяла логическа част от шаблона (вече използвахме това в някои от примерите).

Друга много важна функция на групите е, че те ни дават възможност да извличаме точно тези части от съвпадението, които ни интересуват. Подниозовете, които съвпадат с частите от шаблона в групите, се запазват паралелно с цялото съвпадение. Така можем с едно съвпадение на поголям шаблон едновременно да извлечем различни части текстова информация и с помощта на групирането да ги обособим и да ги използваме поотделно.

Може би най-голямото предимство на групите, що се отнася до процеса на търсене, е възможността да се реализират т. нар. **обратни препратки (backreferences)**. С тяхна помощ можем в хода на изпълнението да получим стойността на съвпадението с дадена група и да я използваме по някакъв начин на друго място по-нататък в шаблона или за заместване, което ни осигурява голяма гъвкавост. Можем например да търсим отварящ HTML таг от вида **`<таг>`** и после да искаме да намерим съответния затва-

рящ – **</tar>**. Ако тук обособим "**tar**" в група, ще можем да търсим затварящия таг именно като използваме обратна препратка към тази група.

Ще разгледаме още някои подробности по синтаксиса на групирането, както и някои примери, които ще демонстрират нагледно изложеното по-горе, след което ще обясним как групите могат да се използват при програмирането с регулярни изрази на .NET.

Неименувани (анонимни) групи

Както вече споменахме, групирането става по най-естествения начин – чрез символите **(** и **)**. Всяка част от шаблона, оградена с кръгли скоби, автоматично се третира от машината на регулярните изрази като група. Когато тази група съвпадне с подниз в текста, машината продължава да търси съвпадения със следващия символ, но запазва в паметта стойността на съвпадението с групата. Тези запазени стойности се индексират с номера, които се запълват в реда на срещането на групите в шаблона.

Анонимни групи – пример

Да видим един прост пример:

Шаблон: **^(\w+)=(\w+);$**

Нека имаме един списък от опции и техните стойности, записани като редове във вида "**option=value;**". Един кратък подобен списък може да изглежда например така:

```
filtering=anisotropic;
details=hi;
resolution=1024;
enable_shades=1;
```

Ако искаме да извлечем двойките опции и стойности, можем да търсим в многоредов режим с помощта на горния израз. На всеки ред първото **\w+** ще открие като съвпадение точно текста на **option**, а второто – текста на **value**. При това, понеже сме поставили тези части от шаблона в скоби, съвпаденията с тях ще бъдат запазени в паметта – първата под номер 1, а втората – под номер 2. Под номер 0 се запазва съвпадението с целия шаблон. На края на реда машината ще спре да търси, защото сме намерили съвпадение. В този момент трябва да извлечем съвпаденията, които сме получили в групите, защото когато поискаме ново търсене, стойностите им ще се презапишат от новите съвпадения.

Това е важно да се запомни – стойността, прихваната от групата, е само тази от последното съвпадение и тя презаписва всички предишни получени стойности. Следният пример демонстрира това:

Текст: **бира**

Шаблон: **`([рбиа])+`**
Запазена в групата стойност: **`а`**

Шаблон: **`([рбиа]+)`**
Запазена в групата стойност: **`бира`**

Тук и в двата случая **+** е "лаком" и съвпадението с шаблона е целият низ. В първия случай това означава четири пъти да се намери съвпадение с частта в групата, защото плюсът се отнася към нея. От тях само последната стойност ("а") остава запазена. Във втория случай групата огражда шаблона и запазва цялото съвпадение.

Именувани групи

Синтаксисът на регулярните изрази в .NET ни дава възможност да слагаме и имена на групите, които дефинираме. Това придава по-добър вид на регулярния израз и обособява по-добре използваните групи като логически цялости, защото става ясно каква е целта им.

Именуваните групи се дефинират подобно на неименуваните, но след отварящата скоба трябва да напишем и името на групата по следния начин: **`(?<name>regular_expression)`** или **`(?'name'regular_expression)`**. Между двата варианта няма разлика – може да се използват като взаимозаменяеми. Първият е по-удобен за низове, защото там трябва да се escape-ват кавичките, докато вторият може да е от полза при ASP тагове, където счупените скоби имат специално значение. Да обърнем внимание, че тук не може да стане объркване с метасимвола **`?`** за 0 или 1 повторения, защото той трябва да стои след валиден за регулярния израз символ или конструкция, а отварящата скоба не е такава.

За яснота, да променим горния шаблон като добавим имена на групите:

Шаблон: **`^(?<option>\w+)=(?<value>\w+);$`**

По този начин сме дефинирали групите **`option`** и **`value`** и можем да достъпваме запазените от тях съвпадения по име, което опростява логиката на програмата и четимостта на кода.

Номериране на групите

Именуваните групи също се номерират заедно с неименуваните, въпреки че имат и име. При това машината на регулярните изрази следва следното правило: подред се номерират първо неименуваните групи по реда на срещането им в израза отляво надясно, а след това и именуваните – пак отляво надясно. Това поведение е характерно само за .NET и не важи за други платформи и езици, които позволяват работа с регулярни изрази (при тях всички групи се номерират в реда на срещане).

Например в израза "**(\w+)_(?<group1>\d+)_(\s+)**" групата **(\w+)** ще получи номер 1, групата **(\s+)** – номер 2, а **(\d+)** ще има номер 3 и ще може също да се достъпва с името **group1**. Това понякога е объркващо и трябва да се внимава. По възможност е добре да се използват или само именувани, или само неименувани групи. Разбира се, ако има групи, които използваме само за да прилагаме метасимволи към част от шаблона наведнъж, няма смисъл да им слагаме имена, защото това затруднява четенето на кода. Там можем да използваме специалната конструкция за група, която не запазва съвпадение. За нея ще стане дума по-късно в темата.

Търсене с групи в .NET

Време е да видим как можем лесно да се възползваме от възможностите на групирането в .NET и да извличаме информацията, запазена в групите.

Класовете Group и GroupCollection

В пространството от имена **System.Text.RegularExpressions** групите се представят с класа **Group**. Този клас е доста подобен по функционалност на класа **Match** – и двете представят низ, който при търсенето се е оказал съвпадение с шаблона, само че **Match** пази съвпадението с целия регулярен израз, а **Group** пази съвпадението с някоя част от шаблона, която е дефинирана като група.

Подобно на **Match**, класът **Group** предлага свойства, които описват мястото на съвпадението, стойността му и дължината му – **Index**, **Value** и **Length**. Свойството **Success** ни показва дали съответната група е намерила съвпадение.

Класът **GroupCollection** е просто колекция от обекти на класа **Group**. Както ще видим след малко, обикновено този клас използваме за достъп до колекция от всички групи, дефинирани в даден регулярен израз. Свойствата и методите на **GroupCollection** са подобни на тези на **MatchCollection** и на повечето колекции в .NET.

Как извличаме информацията от групите?

Както вече знаем, след всяко търсене с регулярен израз, получаваме обект от класа **Match**, който описва съвпадението. За да проверим запазените в групите съвпадения, използваме свойството **Groups** на класа **Match**. Това свойство има стойност от тип **GroupCollection** и ни предоставя точно тези съвпадения. Следният пример показва как да извличаме информацията от групите:

```
static void Main()
{
  Regex regex = new Regex(@"^(\w+)=(\w+);$");
```

```
  string text = "filtering=anisotropic;";

  Match match = regex.Match(text);

  while (match.Success)
  {
     Console.Write("\n\n");
     Console.WriteLine(
        "Съвпадение: \"{0}\" - начало {1}, дължина {2}",
        match, match.Index, match.Length);

     Console.WriteLine("Брой групи: " + match.Groups.Count);
     for (int i=0;i<match.Groups.Count;i++)
     {
        Console.WriteLine(
           "Група номер {0}, име \"{1}\"",
           i, regex.GroupNameFromNumber(i));
        Console.WriteLine(
           "\tСтойност: \"{0}\", започва на {1}",
           match.Groups[i].Value, match.Groups[i].Index);
     }
     match = match.NextMatch();
  }
}

/* Output:
   Съвпадение: "filtering=anisotropic;" - начало 0, дължина 22
   Брой групи: 3
   Група номер 0, име "0"
      Стойност: "filtering=anisotropic;", започва на 0
   Група номер 1, име "настройка"
      Стойност: "filtering", започва на 0
   Група номер 2, име "стойност"
      Стойност: "anisotropic", започва на 10
*/
```

От примера се вижда това, за което говорихме по-рано – че под група с номер 0 се пази цялото съвпадение. Всъщност класът **`Match`** наследява **`Group`** и реално **`match.Groups[0]`** е самият обект **`match`**.

Забележете също употребата на метода **`GroupNameFromNumber(int)`** на класа **`Regex`**. Този метод не е статичен и може да се използва само за обект–шаблон. Връща име на група по даден номер (спомнете си как се номерират групите в израза!). Аналогичният метод **`GroupNumberFromName(string)`** пък ни дава номера на група с дадено име в израза, стига да има такава (и -1 в противен случай).

Именуване и номериране

Именуваните групи можем да достъпваме и чрез името им, например **match.Groups["option"]** или **match.Groups["value"]**. Самите имена на групите можем да получим наведнъж с метода **GetGroupNames()**. Номерата, на които те отговарят съответно, се връщат пък от **GetGroupNumbers()**. И двата метода нямат статични варианти и връщат масиви. С тяхна помощ можем да си припомним начина на номериране на групите – първо анонимните, после именуваните:

```
static void Main()
{
  Regex regex = new Regex(
     @"Пример ((\w+)\s(?<named>[руг]+)(пи))*");
   string text = "Пример с групи";
   Match match=regex.Match(text);
   Console.Write("\nИмена на групи:  ");
   foreach (string name in regex.GetGroupNames())
   {
     Console.Write("{0} <::> ", name);
   }
   Console.Write("\n");

   Console.WriteLine("Номера на групи:");
   foreach (int number in regex.GetGroupNumbers())
   {
     Console.WriteLine("{0} - {1} - {2}", number,
        regex.GroupNameFromNumber(number),
        match.Groups[number].Value);
   }
}
/* Output:
   Имена на групи:  0 <::> 1 <::> 2 <::> 3 <::> named <::>
   Номера на групи:
   0 - 0 - Пример с групи
   1 - 1 - с групи
   2 - 2 - с
   3 - 3 - пи
   4 - named - гру
*/
```

Тук именуваната група е дефинирана преди групата **(пи)**, но понеже **(пи)** е анонимна, тя получава по-малък номер. Анонимните в примера се нареждат по реда на отварящите скоби.

Парсване на лог – пример

Ще разгледаме два по-практически примера за демонстрация на извличането на различни части информация от текста с помощта на групи. Пър-

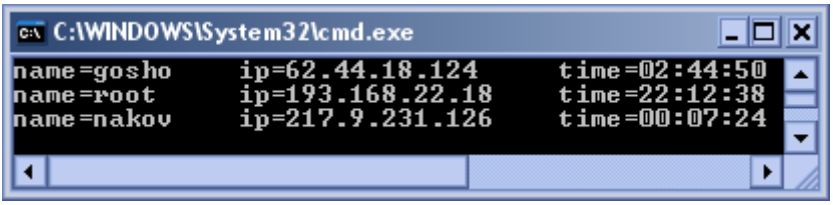
вият пример е за парсване на потребителски лог, в който имаме данни в следния формат:

```
<име на потребител> <IP адрес> <време в системата>
```

Искаме да извлечем и изведем тази информация в малко по-удобен вид. За целта ще използваме шаблон с групи. Ето и целият код:

```
static void Main()
{
  string text = "gosho 62.44.18.124 02:44:50\n" +
                "root 193.168.22.18 22:12:38\n" +
                "nakov 217.9.231.126 00:07:24";
   string pattern = @"(?<name>\S+)\s+" +
                    @"(?<ip>([0-9]{1,3}\.){3}[0-9]{1,3})\s+" +
                    @"(?<time>([0-9]+:){2}[0-9]+)";
   MatchCollection matches = Regex.Matches(text, pattern);
   foreach (Match match in matches)
   {
      Console.WriteLine("name={0,-8} ip={1,-16} time={2}",
      match.Groups["name"], match.Groups["ip"],
      match.Groups["time"]);
   }
}
```

Регулярният израз тук не е съвсем изпипан, но върши работа за примера. В резултат на изпълнението получаваме исканите данни, подредени таблично:



Извличане на хипервръзки в HTML документ – пример

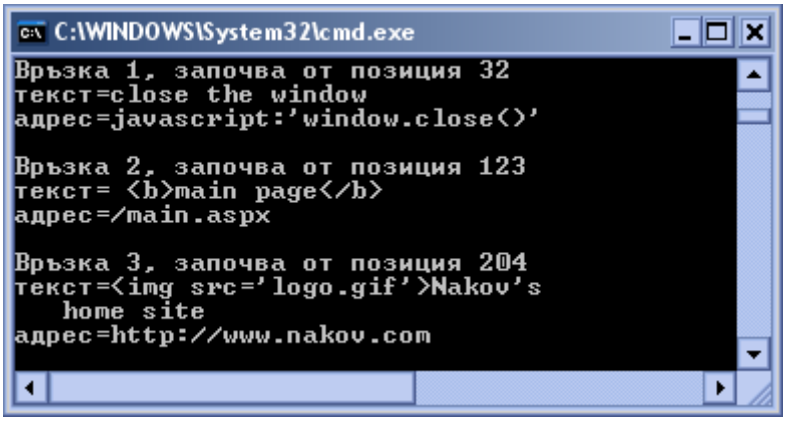
С помощта на групите можем да разширим нашия пример за таговете за хипервръзки, така че да не извличаме целия таг, а само текста му и адреса, към който връзката сочи. За целта изпълняваме следните стъпки.

1. Отваряме VS.NET и създаваме нов конзолен проект.
2. Добавяме връзка към **System.Text.RegularExpressions** в клаузите за пространствата от имена и въвеждаме следния код в главната функция:

```
static void Main()
```

```
{
  string text = @"<html> This is a hyperlink:
   <a href=""javascript:'window.close()'"">close the window</a>
   <br> ... and one more link: <a target=""_blank""
   href=/main.aspx class='link'> <b>main page</b> </a>
   < a href = 'http://www.nakov.com'><img src='logo.gif'>Nakov's
   home site < /a >";

  string hrefPattern = @"<\s*a\s[^>]*?\bhref\s*=\s*" +
       @"('(?<url>[^']*)'|""(?<url>[^""]*)""|" +
       @"(?<url>\S*))[^>]*>" +
       @"(?<linktext>(.|\s)*?)<\s*/a\s*>";

  Match match = Regex.Match(text, hrefPattern);
  int i=1;
  while (match.Success)
  {
     Console.WriteLine("Връзка {0}, започва от позиция {1}",
        i, match.Index);
     string linktext = match.Groups["linktext"].Value;
     Console.WriteLine("текст={0}", linktext);
     string url = match.Groups["url"].Value;
     Console.WriteLine("адрес={0}", url);
     Console.WriteLine();
     i++;
     match = match.NextMatch();
  }
}
```

3. Стартираме програмата и получаваме следния резултат:



Работа с обратни препратки

Стигаме и до по-интересните възможности, които групирането предлага – използването на обратни препратки в регулярния израз. Чрез тях можем да използваме запазеното в групите като част от остатъка от шаблона.

Както вече споменахме, това е особено полезно, ако искаме някаква част от текста да се повтаря, но не знаем точно каква е тя. В текста на шаблона обратните препратки се обозначават с конструкции от вида **\X**, където **X** е номерът на неименуваната група. В примера с настройките и стойностите стойността на **option** можем да използваме в регулярния израз, като напишем **\1**:

```
Шаблон: ^(\w+)=(\1\w+);$
Текст:
filtering=anisotropic;
details=hi;
background=background-image;
resolution=1024;
Съвпадение: background=background_image;
Група 0: background=background_image; (цялото съвпадение)
Група 1: background
Група 2: background_image
```

За разлика от стария пример, тук поставяме условие втората група да търси съвпадение, започващо с подниза, който е вече запазен в първата. Затова съвпадение има чак на третия ред, където началото на **value** в двойката е именно съдържащото се в частта **option**.

Разбира се обратна препратка не можем да използваме в групата, която я дефинира, т.е. не може да имаме например "**\d(\w+\1)$**" като регулярен израз. Това предизвиква неуспешно съвпадение във всеки текст. В частност, метасимволът **\0** не може да се използва никъде в израза като обратна препратка.

Не можем също да използваме нито скобите за групиране, нито обратни препратки вътре в клас от символи с квадратни скоби. Там те губят специалното си значение и стават литерали, като конструкцията **\X** може да означава осмичен ASCII код, както вече видяхме в частта за escaping.

Обратни препратки към именувани групи

В текста на регулярния израз обратна препратка към именувана група става чрез конструкцията **\k<name>** или **\k'name'**:

```
Шаблон: ^(?<option>\w+)=(?<value>\k<option>\w+);$
е еквивалентно на: ^(?<option>\w+)=(?<value>\1\w+);$
```

Както си спомняме, групата **option** е и група номер 1, ето защо двата записа са еквивалентни. Възможно е да използваме и само **\<option>** вместо **\k<option>**. Със следния пример ще търсим в декларации на потребителски имена и съответни пароли и ще извлечем всички редове, при които името и паролата съвпадат:

```
string text =
   "gosho &boza!!36\n" +
   "pesho pesho\n" +
   "ivo kaka*mara\n" +
   "kaka #k@k@22\n" +
   "test test";
string pattern = @"^(?<user>\S+)\s+(\<user>)$";
MatchCollection matches =
   Regex.Matches(text, pattern, RegexOptions.Multiline);
foreach (Match match in matches)
{
   Console.Write("{0} ", match.Groups["user"]);
}
// Output: pesho test
```

Извличане на HTML тагове от документ – пример

С помощта на обратните препратки вече можем да усъвършенстваме по-сериозно нашия пример с таговете за хипервръзки. Този път ще извличаме информация за всички HTML тагове, които срещнем в документа. Това са поднизове от вида: **<tag attr1 attr2 … attrN>text</tag>**. За целта ще използваме следния израз:

```
<\s*(?<tag>[A-Za-z]\w*)(?<attributes>[^>]*)>(?<text>.*?)</\1>
```

Да разгледаме този шаблон внимателно, за да разберем защо той изпълнява поставената задача:

Шаблон: **<\s***
Коментар: Започваме със символа "<" и преминава през празното пространство след него (ако има).

Шаблон: **(?<tag>[A-Za-z]\w*)**
Коментар: Тук очакваме да намерим валиден HTML таг. Нито един такъв таг не започва с цифри, ето защо задължаваме първият символ да е буква, след което следват един или повече word characters. Така намереното име на таг запазваме в групата **tag**.

Шаблон: **(?<attributes>[^>]*)>**
Коментар: Тъй като * от предишната част е "лаком", то тук започваме от границата на нова дума. Отбелязали сме произволен брой символи, различни от **>**, които ни дават подниза с атрибутите на тага – запазваме ги в група **attributes**. Следва и затварящата скоба **>**.

Шаблон: **(?<text>.*?)**
Коментар: Между отварящия и затварящия таг има произволен текст, който ще пазим в групата **text**. Правим звездичката "мързелива", за да

не улови целия низ до края.

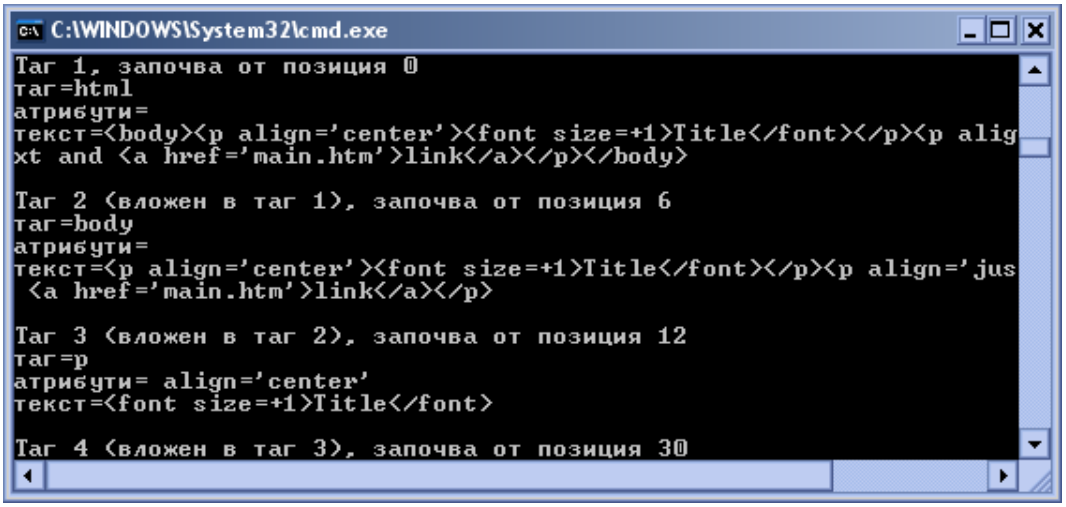
Шаблон: **</\1>**
Коментар: Затварящият таг е същият като отварящия, само че преди името му има символа **/**. Ние запазихме името в групата **tag**, която е също с номер 1. Следователно **\1** ще накара машината на регулярните изрази да търси за точно това име и ще сме сигурни, че сме намерили правилния затварящ таг. Тук можем да използваме и конструкцията **</\s*\1\s*>**, за да улавяме и тагове с празни места вътре.

Нека сега съставим нов проект и въведем кода, който обработва този регулярен израз:

```
using System;
using System.Text.RegularExpressions;

class TagMatch
{
   static void Main()
   {
      string text = "<html><body>" +
         "<p align='center'><font size=+1>Title</font></p>" +
         "<p align='justify'>Text and" +
         "<a href='main.htm'>link</a></p>" +
         "</body></html>";
      string tagPattern = @"<\s*(?<tag>[A-Za-z]\w*)" +
         @"(?<attributes>[^>]*)>(?<text>.*?)</\1>";
      Regex regex = new Regex(tagPattern);
      RecursiveMatch(regex, text, 0, 0, 0);
   }

   static void RecursiveMatch(Regex aRegex, string aText,
      int aTagNumber, int aParentNumber, int aStartIndex)
   {
      MatchCollection matches = aRegex.Matches(aText);
      string outerTagInfo = "";
      if (aParentNumber != 0)
      {
         outerTagInfo = " (вложен в таг " + aParentNumber + ")";
      }
      foreach (Match match in matches)
      {
         aTagNumber++;
         Console.WriteLine("\nТаг {0}{1}, започва от " +
            "позиция {2}",   aTagNumber, outerTagInfo,
            match.Index+aStartIndex);
         string tag = match.Groups["tag"].Value;
         Console.WriteLine("таг={0}", tag);
         string attributes = match.Groups["attributes"].Value;
```

```
        Console.WriteLine("атрибути={0}", attributes);
        string tagtext = match.Groups["text"].Value;
        Console.WriteLine("текст={0}", tagtext);
        RecursiveMatch(aRegex, tagtext, aTagNumber, aTagNumber,
          match.Groups["text"].Index+aStartIndex);
      }
    }
}
```

Налага се да използваме рекурсивно търсене с регулярни изрази заради правилото "всяко следващо търсене започва от края на следващото". Ако не използваме рекурсия, можем да хванем единствено таговете, които не са вложени в други тагове. Ето защо за всеки намерен таг търсим отново рекурсивно в текста, ограден от отварящата и затварящата му част, за да открием вложени тагове. Параметрите на рекурсивната функция са ни нужни, за да изведем правилно информацията. Резултатът от изпълнението на програмата е следният:



Работа с Captures

Класовете **Capture** и **CaptureCollection** ни дават възможност да проверяваме стойностите на всички съвпадения, през които някоя група в регулярния израз е минала в процеса на търсене. Да си припомни, че стойността, която се запазва в групата накрая, е последното съвпадение на групата. Това е от значение например за групи, след които стои количествен метасимвол за повторения – там в групата ще се запази само стойността на последното повторение на шаблона, но не и на предните.

Класът **Capture** е подобен на класа **Group** и класа **Match** и всъщност те са негови наследници. Той представя съвпадение, получено с група в шаблона, но не задължително последното, а което и да е. Стандартните вече свойства **Value**, **Index** и **Length** служат за описване на съвпадението.

Класът **Capture** също няма конструктор. Обекти от този клас получаваме като итерираме колекцията от тип **CaptureCollection**, която се получава като стойност на свойството **Captures** на класа **Group** и класа **Match**.

Следният пример демонстрира употребата на **Captures**:

```
string text = "бира";
string pattern = @"([рбиа])+";
Match match = Regex.Match(text, pattern);
while (match.Success)
{
   Console.WriteLine(
      "\nСъвпадение: \"{0}\" - начало {1}, дължина {2}",
      match, match.Index, match.Length);
   for (int i=0;i<match.Groups.Count;i++)
   {
      Console.WriteLine(
         "Група номер {0}, Стойност: \"{1}\", започва на {2}",
         i, match.Groups[i].Value, match.Groups[i].Index);
      CaptureCollection cc=match.Groups[i].Captures;
      for (int j=0;j<cc.Count;j++)
      {
         Console.WriteLine("\tCapture {0}: \"{1}\", започва" +
            " на {2}",j, cc[j].Value, cc[j].Index);
      }
   }
   match = match.NextMatch();
}
/* Output:
   Съвпадение: "бира" - начало 0, дължина 4
   Група номер 0, Стойност: "бира", започва на 0
        Capture 0: "бира", започва на 0
   Група номер 1, Стойност: "а", започва на 3
        Capture 0: "б", започва на 0
        Capture 1: "и", започва на 1
        Capture 2: "р", започва на 2
        Capture 3: "а", започва на 3
*/
```

Това е примерът, който вече разгледахме при описанието на групите, където отбелязахме, че в групата се запазва само последният съвпадащ с шаблона низ. Всъщност, понеже класът **Group** наследява **Capture**, а в групата се пази именно последният уловен низ, на практика обръщението **match.Groups[i].Captures[match.Groups[i].Captures.Count-1]** ни дава самия обект **match.Groups[i]**. За нулевата група, това съвпада и със самото **match**.

Валидация с регулярни изрази

Валидацията на потребителски вход е едно от най-честите приложения на регулярните изрази в практиката. От гледна точка на сигурността всеки низ, получен след вход от потребителя, е опасен. Той може да не отговаря на очакваните от обработващата програма условия, да не е в същия формат или да надвишава максимално допустимия размер и т.н. – това са проблеми, които могат да доведат до непредвидими действия на програмата, грешни резултати, сривове в системата. Възможно е потребителският вход нарочно да се стреми да постигне това. Много успешни атаки срещу информационни системи, програми или сайтове се дължат именно на недостатъчно валидиране на потребителския вход.

Регулярните изрази са едно изключително удобно средство за такава валидация. Те предоставят лесни за използване методи с много възможности за различни шаблони за проверка, което спестява много програмистки труд по парсване, проверяване и обработване на потребителския вход.

Видове валидация

Обикновено се използват два основни подхода при валидацията на данни. Единият подход е негативен – търси се за неподходящи елементи от данните и ако се намери резултат, низът бива отхвърлен като невалиден. Другият подход е позитивната валидация – да се установят точно условията, на които трябва да отговарят данните и да се провери дали низът ги изпълнява.

Практиката показва, че вторият подход дава по-добри резултати. Причината за това е проста – за програмиста е трудно да предвиди всички възможни типове невалидни данни, които би могъл да получи. Нещо повече, често се случва той да си мисли, че е така, но това да е само фалшива сигурност и програмата му пак да е податлива на пробив. Позитивният подход е по-удобен и сигурен, защото той точно описва какво се очаква като данни от обработващата програма.

Полезни съвета за валидация с регулярни изрази

Следните прости правила помагат при първите опити за валидиране с шаблони:

- Ако се използва негативна валидация, процедурата е подобна на общото търсене с регулярни изрази. Внимание трябва да се отдели на конструирането на израза – обикновено се търси за една от няколко възможности и се използват класове от символи и алтернативен избор, но понякога проверката може да е от различен характер. Например може да проверяваме дали даден елемент се среща повече от 5 пъти последователно и това да са невалидни данни за програма, която очаква всеки елемент да се среща най-много 4 пъти.

- Ако използваме позитивна валидация, трябва да внимаваме за това как точно ще влияят символи като *****, **+**, ***?** и т.н. на шаблона. Около тези символи се правят много грешки, защото не винаги е лесно да се предвиди в даден пример как се държи машината за регулярните изрази, а при валидацията не можем да допуснем възможността някой пример да се окаже изключение от иначе правилния ни шаблон.
- При позитивна валидация винаги трябва да слагаме символите **^** и **$** съответно в началото и в края на шаблона. Това е абсолютно необходимо, защото в противен случай ние валидираме някакъв произволен подниз от данните, но не и целите данни. Дори когато валидацията изисква някакво съвпадение просто да присъства в низа и няма значение с какво е оградено, може поне в началото да използваме конструкция от типа на **^.*pattern.*$**, отколкото само **pattern**, просто за да запомним добре правилото. Това иначе не е много добра практика, защото символите за количество като звездичката забавят обработката на израза.

Валидация с метода IsMatch(...)

Нищо не пречи валидацията да извършваме както и търсенето – с метода **Match(…)** (методът **Matches(…)** обикновено няма смисъл при валидация, защото ние просто проверяваме дали съществува каквото и да е съвпадение). После можем да проверим свойството **Success** на върнатия обект и да валидираме или да отхвърлим данните. Вместо това имаме възможност да използваме и метода **IsMatch(…)**. Той е аналогичен на метода **Match(…)**, но вместо обект от тип **Match**, връща просто булев резултат за успешно или неуспешно съвпадение.

Валидни e-mail адреси – пример

Следният пример показва как можем да проверяваме по-внимателно (в сравнение с примера, който дадохме в началото на темата) за коректно въведени e-mail адреси. Разбира се, той също е сравнително прост и не отговаря напълно на стандарта за адреси (описан в Интернет в RFC 822), но е добра илюстрация за целите на темата. По-сериозен пример, който да се използва и в практиката, може да бъде открит в The Code Project (http://www.codeproject.com/csharp/rfc822validator.asp).

```
string email = Console.In.ReadLine();
string regex = @"^([a-zA-Z0-9_]+(\.[a-zA-Z0-9_\-]){0,49})" +
   @"@(([a-zA-Z0-9][a-zA-Z0-9\-]{0,49}\.)+" +
   @"[a-zA-Z]{2,4})$";
bool valid = Regex.IsMatch(email, regex);
Console.WriteLine(valid);

// Examples of valid emails: diado@kaval.com, kalitko@duduk.net,
// 123--@usa.net, test.test123@en.some-host.12345.com
```

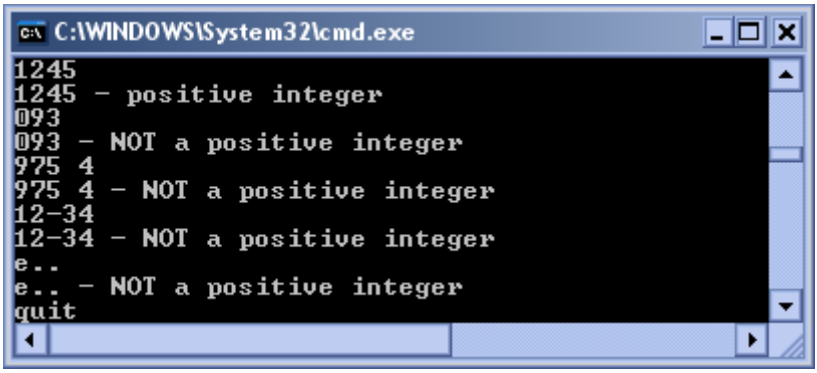
```
// Examples of invalid emails: .ala.@bala.com, test@-host.com,
// user@.test.ru, user@test.ru., alabala@, user@host,
// @eu.net, test%mail.bg
```

Валидни положителни цели числа – пример

Следващият пример показва една възможна проверка за това дали входът, който сме получили представлява положително цяло число (това много често се налага например при попълване на уеб форми):

```
static void Main()
{
   string test = Console.In.ReadLine();
   while (test != "quit")
   {
      Check(test);
      test = Console.In.ReadLine();
   }
}

static bool IsPositiveInteger(string number)
{
   Regex numberRegex = new Regex(@"\A[1-9][0-9]*\Z");
   return numberRegex.IsMatch(number);
}

static void Check(string text)
{
   Console.WriteLine("{0} - {1}", text,
      IsPositiveInteger(text) ? "positive integer" :
      "NOT a positive integer");
}
```

Ето и примерен резултат от изпълнението на горната програма:



Заместване с регулярни изрази

Заместването с помощта на шаблон е още една област в обработката на текст, където работата с регулярни изрази опростява неимоверно обичайната сложна логика за парсване. Регулярните изрази позволяват много гъвкаво да определим какво, къде и с какво да се замести, включително части от оригиналния текст.

Принципът на заместването е прост – първо се търсят съвпадения с шаблона на регулярния израз по стандартния начин (както при метода **`Matches(…)`**). Получените съвпадения се заместват с нови низове, които ние сме определили. В низа за заместването може да има части от съвпадението, от останалия текст, както и произволна обработка над тях, което ни дава на практика неограничени възможности при заместването. Примерите ще изяснят как точно става това.

Заместването с регулярни изрази се осъществява с помощта на метода **`Replace(…)`** на класа **`Regex`**. Този метод доста напомня метода **`Replace(…)`** на класа **`String`**, който вече разгледахме, но именно възможностите, изброени по-горе, както и самото търсене по шаблон, а не по точно определен низ, правят **`Regex.Replace(…)`** много по-функционален метод от съответния му при обикновените низове.

Ако не искаме да заместваме всички съвпадения, можем да укажем това с допълнителен параметър за максимален брой на заместванията.

Заместване със заместващ шаблон

Основната форма на метода **`Replace(…)`** е **`Replace(string text, string replacement)`**. Първият параметър е текстът, в който ще заместваме. Шаблонът, по който търсим фразите за заместване, се пази в **`Regex`** обекта. Методът **`Replace(…)`** може да се вика и статично, като се подава и параметър с регулярния израз.

По-интересен е параметърът **`replacement`**. Това е т.нар. **заместващ шаблон (replacement pattern)**, който се образува по специални правила, подобно на самите регулярни изрази – така се осигурява гъвкавост и се дават много възможности за заместване.

Заместващият шаблон обаче не е регулярен израз. След като специалните символи в него се заместят със съответното им значение, получаваме низ, който стои на мястото на съвпадението в получения низ. Ясно е, че ако заместващият шаблон беше регулярен израз, методът няма да знае с какво да замества – например ако искаме да заместим с **`[абв]`**, то с коя от тези букви всъщност искаме да заместим? Ето защо заместващият шаблон се определя по-други правила, които еднозначно да определят с какво да заместим. Следният пример демонстрира използването на метода **`Replace(…)`**.

Заместване на тагове от форум с HTML – пример

Често в Интернет форумите не се позволява на потребителите да използват HTML в съобщенията си поради опасностите от гледна точка на сигурността. За да се симулира HTML, например за форматиране на текст, картинки или връзки, се използва някакъв вид псевдокод от форум, който после се замества с истински HTML тагове.

В следващия пример имаме тагове от вида **[URL=…]** … **[/URL]**, които искаме да заместим с нормалните тагове за хипервръзки **** … ****:

```
static void Main()
{
   String text = "Here is the link:<br>" +
      "[URL=http://www.devbg.org]БАРС[/URL]<br>\n" +
      "and the logo:[URL=http://www.devbg.org][IMG]\n" +
      "http://www.devbg.org/basd-logo.png[/IMG][/URL]\n";
   string pattern = @"\[URL=(?<url>[^\]]+)\]" +
      @"(?<content>(.|\s)*?)\[/URL\]";
   string newPatt = "<a href=\"${url}\">${content}</a>";
   string newText =
      Regex.Replace(text, pattern, newPatt);
   Console.WriteLine(newText);
}
```

Ако изпълним горната програма, ще видим в изхода всички **[URL]** тагове заместени със съответните им **<a href>** тагове. Това постигаме, като извлечем с помощта на именувани групи адреса (групата **url**) и текста на форумния таг (групата **content**) и после използваме уловените от тях стойности в заместващия шаблон – това става с конструкциите **${url}** и **${content}**. Обикновено точно съвпаденията на групите се използват при заместването, но има още няколко възможности, които можем да използваме, и които накратко ще покажем.

Специални символи в заместващия шаблон

За да демонстрираме как работят специалните символи, ще променим малко горния пример:

```
static void Main()
{
   String text = "before[URL=address]text[/URL]after";
   string pattern = @"\[URL=(?<url>[^\]]+)\]" +
      @"(?<content>(.|\s)*?)\[/URL\]";
   string newPatt = "<a href=\"${url}\">${content}</a>";
   string newText =
      Regex.Replace(text, pattern, newPatt);
   Console.WriteLine(newText);
}
```

и ще наблюдаваме как се изменя изходът при използването на различни конструкции в заместващия шаблон след "**href**":

Конструкция: **$номер** или **${номер}**
С какво замества: с уловеното от групата с указания номер
В шаблона: **${content}**
Изход: **beforetextafter** (защото групата **(.|\s)** е неименувана и е първа в номерирането, а последното уловено от нея е именно **t**)

Конструкция: **${име}**
С какво замества: с уловеното от групата с това име
В шаблона: **${content}**
Изход: **beforetextafter**

Конструкция: **$`**
С какво замества: с частта от входния низ преди съвпадението
В шаблона: **${content}**
Изход: **beforetextafter**

Конструкция: **$'**
С какво замества: с частта от входния низ след съвпадението
В шаблона: **${content}**
Изход: **beforetextafter**

Конструкция: **$&**
С какво замества: с цялото съвпадение
В шаблона: **${content}**
Изход: **beforetextafter**

Конструкция: **$+**
С какво замества: с уловеното от последната група
В шаблона: **${content}**
Изход: **beforetextafter**

Конструкция: **$_**
С какво замества: с целия входен низ
В шаблона: **${content}**
Изход: **before<a**
href="="before[URL=address]text[/URL]after">textafter

Конструкция: **$$**
С какво замества: escape на самото $
В шаблона: **${content}**
Изход: **beforetextafter**

Заместване с MatchEvaluator

Заместващият шаблон без съмнение осигурява доста гъвкавост и възможности при заместването. Понякога обаче се налага да реализираме и по-сложна логика. Тогава можем да използваме метода **Replace(string text, MatchEvaluator evaluator)**. При този вариант като втори параметър подаваме функция, съответстваща на делегата **MatchEvaluator**. Тази функция получава като входен аргумент откритото съвпадение (като обект от клас **Match**) и връща текста, с който то трябва да се замести в изхода (като **string**). В тялото на функцията реализираме логиката, която искаме да обработи съвпадението и да подбере правилно заместващ низ.

Функцията, сочена от делегата, се извиква за всяко открито съвпадение при заместването. За пример ще разгледаме как можем с помощта на **MatchEvaluator** да направим първите букви на всички думи в даден текст главни:

```
static string CapitalizeFirstLetter(Match match)
{
   string word = match.Value;
   return Char.ToUpper(word[0]) + word.Substring(1);
}

static void Main()
{
   String text = "бирено парти - вход свободен!";
   string pattern = @"\w+";
   string newText = Regex.Replace(text, pattern,
     new MatchEvaluator(CapitalizeFirstLetter));
   Console.WriteLine(newText);
}
// Output: Бирено Парти - Вход Свободен!
```

Разделяне на низ по регулярен израз

По регулярен израз може и да се разделя низ, подобно на метода **Split(…)** при класа **String**. Класът **Regex** също има метод **Split(…)**, който обаче може да търси за разделителя чрез шаблон, което разбира се прави процедурата по разделянето по-лесна и по-функционална.

Методът има сигнатура **string[] Split(string text)** или в статичния вариант **string[] Split(string text, string separator)**, където **separator** е регулярният израз, по който ще разделяме. При първия вариант той се пази в обекта. Търси се последователно и наведнъж, както при **Replace(…)** и всяко съвпадение с шаблона се приема за разделител. Може да се използват допълнителни параметри за максимален брой разделяния и за начална позиция на търсене.

Ще покажем как с помощта на **`Split(…)`** можем да извлечем e-mail адреси от списък с разнообразни разделители:

```
static void Main()
{
   String text = "kaka@kaval.bg;; pesho@magurele.net, " +
      "bob@mail.bg\n\nfn12345@fmi.uni-sofia.bg\n" +
      "    mente@eu.int | , , ;;; gero@dir.bg";
   string splitPattern = @"[;|,|\s|\|]+";
   string[] emails = Regex.Split(text, splitPattern);
   Console.WriteLine(String.Join(", ", emails));

   // Output: kaka@kaval.bg, pesho@magurele.net, bob@mail.bg,
   // fn12345@fmi.uni-sofia.bg, mente@eu.int, gero@dir.bg
}
```

В случай, че регулярният израз приема за съвпадение празния низ, трябва да внимаваме за неочаквани резултати, например:

```
static void Main()
{
   string text = "text";
   string splitPatternGreedy = @"\w*";
   string[] result = Regex.Split(text, splitPatternGreedy);
   Console.WriteLine(String.Join(", ", result));

// Output: , ,

  string splitPatternLazy = @"\w*?";
   result = Regex.Split(text, splitPatternLazy);
   Console.WriteLine(String.Join(", ", result));

// Output: , t, e, x, t,
}
```

В първия случай цялата дума **`text`** става разделител, защото звездата е "лакома", и в масива получаваме два пъти празния низ – веднъж от мястото в началото на текста и веднъж от края. При втория случай празният низ е винаги разделител, защото звездата е "мързелива", и в масива получаваме празните низове от началото и от края, както и всеки символ от текста поотделно.

Разделяне с групи в шаблона

Ако в шаблона за разделителя има обособени групи, то те участват като низове в резултата. Следният пример демонстрира това:

```
static void Main()
{
   string[] parts = Regex.Split("скара - бира", @"\s*(-)\s*");
```

```
   Console.WriteLine(String.Join("::", parts));
   // Output: скара::-::бира
}
```

Методите Escape(...) и Unescape(...)

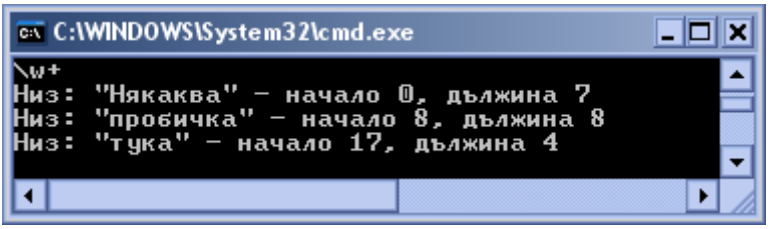
Когато сглобяваме регулярен израз динамично, естествено е да възникнат проблеми, ако не внимаваме за правилния escaping на специалните символи. В подобни случаи е полезно да се използва статичният метод **`Escape(string text)`**. Той преработва низа **`text`**, така че всички метасимволи от синтаксиса на регулярните изрази се escape-ват. След това можем да го използваме като низ от литерали в шаблони.

Нека например искаме да проверим дали дума, въведена от потребителя, се среща в даден текст:

```
string text = @"Някаква пробичка тука";
string word = Console.ReadLine();
string pattern = @"\b" + Regex.Escape(word) + @"\b";
Match match = Regex.Match(text, pattern);
while (match.Success)
{
   Console.WriteLine(
      "Низ: \"{0}\" - начало {1}, дължина {2}",
      match, match.Index, match.Length);
   match = match.NextMatch();
}
```

Шаблона образуваме, като прочитаме потребителския вход и прилагаме върху него метода **`Escape(…)`**, като ограждаме полученото с метасимволите **`\b`**, за да сме сигурни, че ще търсим точно цели думи, а не произволни поднизове.

Ако сега въведем на входа например **`\w+`** няма да получим нито едно съвпадение. Нека след това опитаме да заменим **`Regex.Escape(word)`** само с **`word`** и отново въведем **`\w+`**. Тогава, понеже въведеният метасимвол запазва специалното си значение, резултатът е следният:



В горния код все пак има известна несигурност. Проверете какво би станало, ако потребителят въведе празен низ и дали това е коректно поведение на програмата.

Забележете, че **Escape(…)** слага знака **** и пред празни пространства (заради коментарите в регулярните изрази, за които ще говорим по-късно), което понякога изглежда неочаквано.

Винаги, когато конструирате динамично регулярен израз, прилагайте escaping за низовете, които използвате, особено ако те идват от външен източник.

Ако не прилагаме escaping в примери като горния, в които динамично сглобяваме шаблон за търсене, освен некоректно поведение на търсенето, можем да си доведем и други проблеми. Възможно е потребителят да въведе такъв регулярен израз, че приложението да "зависне". Ще дадем пример за това в секцията за ефективността не регулярните изрази.

Методът Unescape(...)

Методът **Unescape(…)** обръща ефекта на **Escape(…)**. Той конвертира escape последователностите в низа обратно до съответните им символи. При **Unescape(…)** това важи за всички изброени по-рано в темата такива последователности. Например поднизът "**\x6b**" се конвертира до "**k**". По тази причина **Escape(…)** и **Unescape(…)** не са точно противоположни, защото **Escape(…)** няма да промени "**k**" в "**\x6b**".

Настройки и опции при работа с регулярните изрази

На няколко пъти вече споменавахме възможността за допълнителни настройки при работата с регулярни изрази. Това се постига с помощта на свойството **Options** на класа **Regex**. То приема стойности от изброения тип **RegexOptions**, които могат да се обединяват чрез побитовото "или", което означаваме с **|**.

Свойството **Options** е само за четене и не можем да задаваме директно стойността му. Това обикновено правим още в конструктора на **Regex**, например:

```
Regex regex = new Regex(@".*шаблон.*", RegexOptions.IgnoreCase);
```

Параметър от тип **RegexOptions** можем да подаваме и при статичните методи на класа **Regex**, като разбира се в този случай опциите важат само за търсенето, което в момента изпълняваме:

```
Match match = Regex.Match("текст с шаблон",@"(\*)шаблон(.*)",
   RegexOptions.ExplicitCapture | RegexOptions.IgnoreCase);
```

Ще разгледаме по-важните опции, които са на разположение за работа с регулярните изрази. Пълният списък членове на изброения тип **RegexOptions** може да бъде намерен в MSDN.

Multiline

Опцията **Multiline** променя значението на символите **^** и **$**. Както обяснихме в частта за синтаксиса, когато тази опция е активирана, тези символи (но не **\A** и **\Z**) намират съвпадение не само в началото и края на низа, но и в началото и края на всеки ред:

```
string text = "Бирата намаля.\nДайте още бира!";
string pattern = @"^\w+";
MatchCollection matches = Regex.Matches(
    text, pattern, RegexOptions.Multiline);
foreach (Match match in matches)
    Console.Write("{0} ", match);
//Output: Бирата Дайте
```

Singleline

Тази опция променя значението на символа точка. Когато тя е активирана, той намира съвпадение с всеки символ, включително и символа за нов ред **\n**. Без опцията новият ред не е съвпадение с точката:

```
string text = "Бирата намаля.\nДайте още бира!";
string pattern = @".*";

Match match = Regex.Match(text, pattern);
Console.WriteLine("{0}", match);

match = Regex.Match(text, pattern, RegexOptions.Singleline);
Console.WriteLine("\n{0}", match);

/* Output:
Бирата намаля.

Бирата намаля.
Дайте още бира!
*/
```

IgnoreCase

Търсенето с регулярни изрази обикновено разпознава главни и малки букви, но с помощта на опцията **IgnoreCase** можем да извършваме **case-insensitive search**:

```
string text = "Бирата намаля. Дайте още бира!";
string pattern = @"\bбир\w*\b";
MatchCollection matches = Regex.Matches(
   text, pattern, RegexOptions.IgnoreCase);
foreach (Match match in matches)
   Console.Write("{0} ", match);
```

```
// Output: Бирата бира
```

ExplicitCapture

Опцията **ExplicitCapture** задава ново поведение на механизма на групите. Когато тя е включена, неименуваните групи не запазват съвпадение. Ние пак можем да ги използваме за да групираме логически части от шаблона и да прилагаме различни метасимволи към цели групи от символи, но съвпаденията, които те улавят, не се запазват и не можем да ги използваме нито чрез свойството **Groups**, нито като обратни препратки в тялото на израза. Съвпадение запазват единствено групите, за които сме указали име:

```
string text = "Бирата намаля. Дайте още бира!";
string pattern = @"(?<sentence>\w+(\s+\w+)*(\.|!|\?))";
MatchCollection matches = Regex.Matches(
  text, pattern, RegexOptions.ExplicitCapture);
for (int i=0;i<matches.Count;i++)
{
  Console.WriteLine("\nMatch {0}\n",i);
  foreach (Group group in matches[i].Groups)
    Console.WriteLine("{0}", group);
}
/* Output:
Match 0

Бирата намаля. //Group 0 - entire match
Бирата намаля. //Group 1 - sentence

Match 1

Дайте още бира! //Group 0 - entire match
Дайте още бира! //Group 1 - sentence
*/
```

Това поведение е удобно, когато имаме нужда да групираме части от израза, но запазеното в тях не ни трябва и не е нужно да хабим памет за него.

RightToLeft

С тази опция можем да изпълняваме търсенето отзад напред в низа. Срещат се практически случаи, при които това се налага (например търси се последната дума или последната специална дадена конструкция). В следния пример се вижда резултатът от прилагането на опцията:

```
string text = "Бирата намаля. Дайте още бира!";
string pattern = @"\w+";
```

```
Match match = Regex.Match(text, pattern);
Console.WriteLine("{0}", match);
match = Regex.Match(text, pattern, RegexOptions.RightToLeft);
Console.WriteLine("\n{0}", match);
/* Output:
Бирата

бира
*/
```

Compiled

Ако създадем регулярен израз, използвайки тази опция в конструктора, то той се обработва по различен начин от JIT компилатора. Работата с такъв израз е по-бърза, но ресурсите, заети при създаването му не могат да бъдат освободени в хода на програмата.

Допълнителни възможности на синтаксиса на регулярните изрази

Синтаксисът на регулярните изрази предлага още няколко интересни възможности, които бихме могли да използваме. Някои от тях са с доста ограничена употреба, други са полезни, но не толкова лесни за овладяване и изискват известен опит, за да се научим да ги употребяваме правилно. Ще разгледаме накратко по-интересните.

Символът \G – последователни съвпадения

С метасимвола **\G** указваме на машината на регулярните изрази, че искаме следващото съвпадение да започва оттам, откъдето е свършило предното. Ако това е първият опит за търсене, то съвпадение с този специален символ е единствено началото на низа.

Следният пример демонстрира употребата на тази конструкция:

Шаблон: **\G\s?\w+\s?**

Текст: **Каменицата свърши, имаме само Загорка**
Първо съвпадение: **Каменицата**
Второ съвпадение: **свърши**
Трето съвпадение: **няма** (има и други думи, заобиколени с празно място в текста, но от мястото на последното съвпадение започва само запетая, която не отговаря на **\s?\w+\s?** и следователно до края няма други съвпадения)

Текст: **– Каменицата свърши, имаме само Загорка**
Първо съвпадение: **няма** (при първото търсене **\G** съвпада с началото на

низа, но следва тире, което не удовлетворява шаблона, и отново до края на текста вече не може да има друго съвпадение, защото **\G** няма да се удовлетвори никъде)

За разлика от Perl, където позицията на последното съвпадение се пази глобално в специална променлива, в .NET това не е така. Този механизъм е в сила само поотделно за всеки регулярен израз и съответния му обект или статично търсене. Ето защо ако търсим с два регулярни израза последователно, не можем при втория да използваме края на съвпадението, открито от първия в текста, което доста ограничава употребата на конструкцията. Смисълът от използването й е в ситуации, когато искаме да изберем тези срещания на шаблона, които са едно до друго.

Другата особеност на този метасимвол е, че той трябва да се използва само в началото на шаблона. Това е очевидно, тъй като няма начин предишно съвпадение да е свършило по средата на новото – машината започва да търси наново именно от края на последното съвпадение.

Групи, които не запазват съвпадение

Ако не се нуждаем от запазването на съвпадението, открито от групата, можем да използваме специален синтаксис и да забраним това запазване. Например, да предположим, че търсим във файл с данни за служители, в който за всеки служител на един ред са записани различни части лична информация във вида: **<име>-<фамилия>-<град>-<телефон>-<семейно положение>-**… и т.н. Един такъв файл може да изглежда ето така:

```
<Петър>-<Ангелов>-<София>-<029879898>-<женен>
<Сийка>-<Качакова>-<Плевен>-<064758543>-<омъжена>
```

Ако искаме да извлечем всички телефони от този файл, можем да използваме израз от вида:

```
^(<\w+>-){3}(?<phone><\d+>)-(<\w+>-?)+$
```

Понеже знаем, че телефонът е четвърти на реда, прихващаме първите три типа данни с **(<\w+>-){3}**, след което в групата **phone** взимаме нужния ни номер и завършваме с **-(<\w+>-?)+$**, за да прихванем оставащите до края на реда данни и да можем при следващото търсене да започнем успешно на следващия ред (използваме **Matches** и **Multiline**).

В този пример ние всъщност не се интересуваме от стойността на двете групи, които се срещат в израза и са различни от **phone**. Използваме ги просто за да получим искания резултат. В такъв случай е препоръчително да използваме синтаксиса **(?:group)** (да създадем т.нар. **non-capturing group**) и да не запазим уловеното в групите. Ползата от това е, че запазването по принцип бави обработката на израза, а така си спестяваме забавянето. В един по-сложен пример, където обикновено има и търсене с

връщане назад (backtracking), това забавяне може да бъде реален проблем. Затова е по-добре да преработим израза така:

```
^(?:<\w+>-){3}(?<phone><\d+>)-(?:<\w+>-?)$.
```

Същият ефект се постига с вече разгледаната опция **ExplicitCapture**. Синтаксисът **(?:group)** се използва, когато по някаква причина искаме да използваме неименувани групи, които да пазят съвпадение (което не може да стане с опцията).

Метасимволи за преглед напред и назад

Това са няколко конструкции, които незаслужено се радват на твърде малка популярност, може би защото изглеждат сложни за разбиране на пръв поглед. Всъщност те са много удобни за редица задачи, които могат да се решат и с по-прости синтактични средства, но с цената на много по-дълги и объркани регулярни изрази. А има и проблеми, които са нерешими без средствата за преглед напред и назад.

Преглед напред

Конструкцията **(?=expression)** се удовлетворява, ако от текущата позиция в текста следва съвпадение с израза, дефиниран от **expression** (произволен регулярен израз). Това наричаме преглед напред. Той реално е метасимвол за местоположение (zero-width assertion), т.е. след като провери дали може да намери успешно съвпадение, текущата позиция в низа отново се връща там, където е била в началото. Ако има такова съвпадение, проверката по регулярния израз продължава. В противен случай търсенето пропада и няма съвпадение.

За яснота да разгледаме следния пример. Имаме списък от служители във вид на редове от типа "**име-професия**". Искаме да извлечем имената на тези от тях, които са секретарки. Бихме могли да направим това с търсене в многоредов режим и шаблон като "**^[А-Яа-я]+-секретарка$**", т.е. да зададем "**секретарка**" като подниз от литерали, който задължително трябва да се среща. Проблемът тук е, че на нас това "**-секретарка**" всъщност не ни трябва и ще трябва да го режем всеки път от резултата, за да получим това, което ни интересува. Едно решение е да обособим частта с името в група и да използваме само нея, но това пак е по-скоро заобикаляне на проблема, отколкото решаване.

С израза "**^[А-Яа-я]+(?=-секретарка)$**" постигаме точно необходимата функционалност. Ако след името следва низът за секретарка, търсенето успява, в противен случай пропада, а понеже прегледът не участва в самото съвпадение, резултатът е точно необходимото ни име от реда.

Преглед напред с отрицание

Този метасимвол се означава **(?!expression)** и действа по аналогичен начин, но успява тогава, когато **НЕ** се намери последващо съвпадение с

израза в скобите. В противен случай пропада. Това е много удобно, когато искаме да търсим за нещо, което със сигурност не е следвано от нещо друго. Освен това за разлика от позитивното търсене, търсенето с отрицание позволява конструкции, които просто не са възможни без него.

Да разгледаме пак същия пример, но нека сега ни трябват тези служители, които не са секретарки. Ще използваме следния шаблон:

```
^(?<name>[А-Яа-я]+)-(?!секретарка)(?<job>[А-Яа-я]+)$
```

След като обработи името на служителя, търсенето ще пропадне, ако следва поднизът "**секретарка**". За всяка друга професия, машината ще върне текущата позиция след тирето и ще прихване професията в групата **job**. Подобно нещо не можем да постигнем по друг начин и причината е, че синтаксисът не позволява да се търси отрицание на цяла фраза, а само на клас от символи. При положителния преглед напред обикновено можем просто да заместим прегледа със самата фраза (макар че е неудобно, ако искаме да я обработим по по-специален начин или не ни трябва в резултата), но при отрицанието това няма как да стане освен с конструкцията **(?!expression)**.

Преглед назад и преглед назад с отрицание

Аналогично на прегледите напред, синтаксисът позволява да се използва конструкция за преглед назад. Тя е zero-width assertion, която се удовлетворява, когато текущата позиция е предхождана от подниз, отговарящ на търсения шаблон. Можем да дефинираме и преглед назад с отрицание, който пък се удовлетворява, когато преди текущата позиция няма подниз, отговарящ на шаблона (това може да е и началото на низа!).

Синтаксисът на прегледите назад е следният: Изразът **(?<=expression)** се използва за позитивен преглед назад, а **(?<!expression)** – за негативен. За разлика от други платформи и езици, които не се справят с повечето метасимволи при прегледи назад, в .NET можем да използваме произволен регулярен израз в скобите.

Шаблон: **(?<=П[А-Яа-я]+-)[А-Яа-я]+**
Петров-техник – има съвпадение "техник"
Манева-координатор – няма съвпадение (името не започва с "П")

Горният пример показва как можем да вземем професиите на всички служители, чиито имена започват с "П", при това без да пазим излишна информация в съвпадението.

Условен избор

Една от сравнително по-рядко използваните конструкции е тази за условен избор. Тя е малко сложна за употреба, но предлага много интересни

възможности за разклоняване на възможните съвпадения в зависимост от някакво условие. Синтаксисът е следният:

```
(?(if_expr)then_expr|else_expr)
```

Подобно на прегледа напред, машината първо проверява дали може да намери съвпадение с **if_expr**, започвайки от текущата позиция. Това е съвпадение с нулева дължина, т.е. след като свърши проверката, машината се връща обратно на същата позиция. Ако има съвпадение с **if_expr**, машината продължава по шаблона, определен от **then_expr**, а в противен случай – по този, определен от **else_expr**. Else-частта не е задължителна – ако не присъства, то машината просто продължава по шаблона след израза за условен избор.

Разликата с прегледа напред е, че тук можем да разклоним шаблона. Например, нека имаме поредица от имена на градове от по шест букви. Единственото изключение трябва да бъде за градове, започващи с "В", които пък трябва да са от по пет букви.

Шаблон: **\b(?(B)\w{5}|\w{6})\b**
Видин, Дъблин, Монтана, Ванкувър – съвпадения "Видин" и "Дъблин"

Условие с група

Една по-използваема форма на конструкцията за условен избор е следният синтаксис: **(?(group)then|else)**. При нея не се прави проверка за преглед напред, а вместо това се проверява дали дотук е намерено съвпадение за група с име **group** (или с този номер, ако **group** е число).

Шаблон: **((?<mail>From|To)|Subject): ((?(mail)\w+@\w+\.\w+|.+))**
Текст:
From: example@mail.bg
To: nomatter@yahoo.com
Subject: mail
To: nomatter+ok.com
Коментар: Тук има съвпадение на всеки ред, без последния. На първите два групата **mail** е уловила резултат и в условната конструкция има проверка за мейл, която се удовлетворява. На третия няма валиден мейл, но групата **mail** не намира съвпадение и затова шаблонът продължава с **.+**, което се удовлетворява от думата "**mail**". На последния ред има съвпадение в групата, но мейлът не е валиден.

Когато искаме да зададем в условието изрично да се търси по израз, а не по име на група (в случай, че има двусмислица), можем да използваме конструкцията **(?(?=expr)then|else)**.

Коментари в регулярните изрази

Както се убедихме, регулярните изрази обикновено са доста дълги и доста трудни за четене и възприемане. Веднъж написани, после е трудно да бъдат разбрани внимателно отново, особено от друг програмист. Ето защо възможността да се слагат коментари в израза е много полезна и значително опростява задачата по разчитане на шаблона. Препоръчва се да използваме коментари при всеки по-дълъг израз.

Коментари се въвеждат с конструкцията **(?#comment)**. Някои видове софтуер за работа с регулярни изрази поддържат оцветяване на тези конструкции, така че четенето да става още по-лесно. Ето пример за коментар от шаблона за HTML таговете:

```
(?#open)<([A-Za-z]\w*)(?#attr)[^>]*>(?#text).*?(?#close)</\1>
```

Опцията IgnorePatternWhitespace

Въпреки че все пак улесняват разбирането на израза, коментарите доста затрудняват четенето му, защото натоварват конструкцията. Активирайки опцията **RegexOptions.IgnorePatternWhitespace**, можем да използваме и по-прегледни коментари. При тази опция знакът **#** означава коментар до края на реда, а празните пространства се игнорират (и може да се използват за прегледност). Ако искаме да използваме празни пространства и **#** в шаблона, трябва да използваме escaping (методът **Escape(…)** също го прави).

```
<([A-Za-z]\w*)  #begin opening tag
[^>]*           #attributes
>               #end opening tag
.*?             #text in tag
</\1>           #closing tag (matches the opening tag)
```

Модификатори на регулярните изрази

Пет от опциите за регулярни изрази имат съкратени еднобуквени означения, които можем да използваме направо в израза и да ги включваме и изключваме за части от шаблона. Това понякога се оказва доста удобно. Споменатите означения са **i**, **m**, **n**, **x** и **s** – съответно за **IgnoreCase**, **Multiline**, **ExplicitCapture**, **IgnorePatternWhitespace** и **Singleline**. Синтаксисът за употребата им е следният:

- активиране: **(?imnxs)**. Можем да сложим произволна комбинация от петте символа в скобите, например **(?isxn)**. Това означава, че оттук надясно в израза (или докато не бъдат отменени), посочените опции важат.
- дезактивиране: **(?-imnsx)**. Аналогично, дезактивира всички посочени опции оттук надясно. Двете могат да се обединят,

например **(?im-s)**. Всички отляво на минуса се активират, а тези отдясно се дезактивират.

- в рамките на група: Ако искаме да включим или изключим някаква опция само за част от шаблона, можем да използваме този синтаксис: "**(?i)some_part(?-i:case_sensitive_part)rest**", т.е. да дефинираме група, която не пази съвпадение, и да й прикачим съответните модификатори.

Следният пример демонстрира модификатора **i**:

Шаблон: **(?i)би(?-i)ра** или **(?i:би)ра**
бира – съвпадение "бира"
бИра – съвпадение "бИра"
бИрА – няма съвпадение (при "ра" вече нямаме **IgnoreCase**)

Особености и метасимволи, свързани с Unicode

Когато проверяваме за валидност потребителски вход, а и когато извличаме информация с регулярни изрази, трябва да имаме предвид възможността да работим в интернационална среда. В една обикновена форма за попълване с име и фамилия бихме могли да валидираме името с израз от типа на "**^[A-Za-z]$**", но това няма да е коректно за думи като Möller или Jerôme, които са съвсем реални имена.

Всеки, който се е сблъсквал с проблемите на интернационализацията на програмите, се е убедил, че преодоляването им съвсем не е тривиално. Ако работим без Unicode и само с различни кодови таблици, тези проблеми се задълбочават. Но дори и с Unicode не винаги сме в състояние да определим някакви точни граници на валидни за случая Unicode кодове, така че да използваме конструкции като **[\uXXXX-\uYYYY]**. В езиците съществуват множество изключения и специални случаи, които затрудняват обработката – например в английския език по принцип няма ударения, но има думи като "caffé", като тук последната гласна не влиза в стандартните граници на латинските символи в Unicode.

Метасимволите за Unicode категории

Подобни проблеми можем да решаваме лесно чрез специалните метасимволи за категории **\p{category}** и **\P{category}**. В Unicode всеки символ принадлежи на определена категория (например главни букви, десетични арабски цифри, римски цифри, математически символи и т. н.).

Конструкцията **\p{category}** намира съвпадение с всеки Unicode символ, който влиза в категорията **category**. Например "**\p{Ll}**" ще открие всички малки букви, а "**\p{Sc}**" всички символи за валута като **$**, **€** или **¥**.

Обратно, конструкцията **\P{category}** търси за всички символи, които не са в указаната категория, например "**\P{Nd}**", ще намери съвпадение с всички символи, които не са арабски цифри.

Пълен списък с Unicode символите по категории може да бъде открит на http://www.fileformat.info/info/unicode/category/, както и на. Основните категории са: **L (Letters)**, **N (Numbers)**, **M (Marks)**, **P (Punctuation)**, **Symbols (S)**, **Separators (Z)** и **Other (C)**, като всяка от тях си има по няколко подкатегории, например **Pd (Punctuation-Dash)** са всички видове тирета, Sm **(Symbols-Math)** са всякакви математически символи и т.н. Можем да използваме и конструкции като **IsHebrew**, **IsArabic** и т.н., с които можем да проверим дали даден символ е съответно в еврейската или арабската азбука. Например **\p{IsKatakana}** ще намери съвпадение с всички символи от японската азбука Катакана.

Как да преработим шаблона за имената?

Като използваме тези конструкции, с шаблон от вида на **[\p{Ll}\p{Lu}\p{Lo}]+** (Letter-Lowercase, Letter-Uppercase, Letter-Other) можем да направим по-добра и съобразена с интернационалните условия проверка за име. Ще отбележим, че в повечето случаи бихме могли просто да използваме конструкцията **\w+**. Освен ако е специално указано противното със съответната опция, метасимволът **\w** е еквивалентен на alphanumeric клас в термините на Unicode, т.е. включващ символи от всички азбуки. Все пак това не винаги е уместно, защото **\w** е точно еквивалентен на **[\p{Ll}\p{Lu}\p{Lo}\p{Lt}\p{Nd}\p{Pc}]**, което включва числа, символи за свързване на думи като "_" (underscore), заглавни букви и т. н., а тези символи понякога (както и в случая с имената) не са ни нужни.

Възможни проблеми с Unicode

Вътрешната поддръжка на Unicode за низовете в .NET Framework значително намалява проблемите, които се срещат при други платформи с конверсията на символи от други кодови таблици. При все това трябва да внимаваме за някои потенциални опасности.

Да разгледаме символа "**à**". Този символ е една визуална графема, но може да се състои от два различни Unicode символа (точкова двойка). Например той може да се кодира чрез символа на буквата "a" (\u0061) плюс символа за комбиниращо затворено ударение \u3000.

При това възниква проблем с нашите шаблони. Ако буквата е кодирана с точкова двойка от два символа, то изразът **.** (точка), приложен върху **à**, ще намери като съвпадение първо **a**, а после ударението. Цялата графема можем да прихванем с **..** (две точки), а например изразът **^.$** ще се провали, защото символите са два, а не един.

Можем евентуално да предвиждаме подобни ситуации и да съобразяваме нашите шаблони с това поведение на Unicode символите, но за съжаление

символът **à** може да се кодира и с единствен Unicode символ – **\u00E0**. Това е така по исторически причини, понеже този символ присъства в кодовата таблица Windows-1252. Ясно е, че няма откъде да знаем дали буквата, която получаваме на входа, отговаря на един или на два Unicode символа, което усложнява нещата.

Решения на подобни проблеми

В такива случаи отново можем да използваме механизма на категориите. Шаблонът "**\P{M}\p{M}***" решава проблема – той намира за съвпадения както единични символи, които не са комбиниращи знаци (Marks), така и комбинации от единични символи и такива знаци. Тази конструкция можем спокойно да използваме вместо точката, когато ни се налага да решаваме такъв проблем. Това, че символът * е "лаком", не бива да ни притеснява, защото няма как да извлечем по погрешка част от втора графема – всички графеми трябва да започват със знак, който не е комбиниращ символ.

Повече информация относно регулярните изрази и Unicode може да се открие на страницата "Unicode Regular Expression Guidelines" (http://www.unicode.org/reports/tr18/).

Предварително компилиране и запазване на регулярни изрази

Една интересна възможност на библиотеката за регулярни изрази в .NET Framework е компилирането им до самостоятелно асемблии. Чрез статичния метод **Regex.CompileToAssembly(RegexCompilationInfo[],AssemblyName)** ние можем да запазим произволен брой тествани регулярни изрази в DLL модул, чрез който после да ги използваме в нашите приложения.

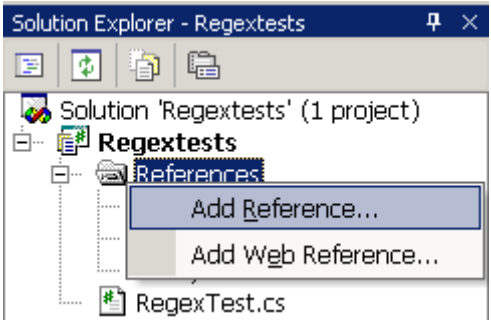
Класът **RegexComilationInfo** е помощен клас, в който пазим информация за шаблона на запазвания регулярен израз, за името, с което ще го достъпваме, за пространството му от имена, както и за настройките от тип **RegexOptions**, с които ще го използваме. Класът **AssemblyName** съхранява име, версия и други данни за асемблито, в което ще запазим регулярните си изрази. Подробна информация и за двата класа може да се намери в MSDN. Ще покажем един кратък пример за използването на тази технология с някои от вече разгледаните регулярни изрази:

```
using System;
using System.Text.RegularExpressions;
using System.Reflection;
using System.IO;

class RegexClasses
{
   static void Main()
   {
```

```
      // Positive integer pattern
      string positiveIntPattern = @"^[1-9][0-9]*$";
      // Unicode name pattern
      string unicodeNamePattern="\b([p{Ll}p{Lu}p{Lo}]p{M}*)+\b";
      // Option=value declarations pattern
      string optionValuePattern = @"^(\w+)=(\w+);$";

      RegexCompilationInfo[] compileInfo = {
         new RegexCompilationInfo(positiveIntPattern,
         RegexOptions.None, "PositiveInteger", "RegexLib", true),
         new RegexCompilationInfo(unicodeNamePattern,
         RegexOptions.None, "UnicodeName", "RegexLib", true),
         new RegexCompilationInfo(optionValuePattern,
         RegexOptions.Multiline, "OptionValueDeclaration",
         "RegexLib", true)
      };

      AssemblyName assemblyName = new AssemblyName();
      assemblyName.Name = "RegexLib";
      assemblyName.CodeBase = Directory.GetCurrentDirectory();
      assemblyName.Version = new Version("1.0.0.0");

      Regex.CompileToAssembly(compileInfo, assemblyName);
   }
}
```

С този код ще запазим генерираната библиотека като DLL файл в текущата директория (това е поддиректорията **\bin\Debug** на директорията, в която е файлът с проекта ни). За да използваме регулярните изрази, които запазихме, трябва в проекта, където са ни необходими, да добавим **референция (Reference)** към новото асембли.

За целта в прозореца **Solution Explorer** на Visual Studio .NET (по подразбиране в дясната част на екрана) намираме **References** в дървото, щракваме с десен бутон и избираме **Add Reference**. Оттам с бутона **Browse** намираме пътя до нашата DLL библиотека и я добавяме.



Сега остава само да добавим и реда "**using RegexLib;**" в кода на проекта и можем да използваме запазените регулярни изрази (класовете

PositiveInteger, **UnicodeName** и **OptionValueDeclaration** в асемблито) подобно на класа **Regex** ето така:

```
using System;
using System.Text.RegularExpressions;
using RegexLib;

class RegexClassUse
{
   static void Main()
   {
      string text = "98760";
      PositiveInteger regex = new PositiveInteger();
      Console.WriteLine(regex.IsMatch(text) ?
         "Success" : "Failure");
      }
   }
}
// Output: Success
```

Разбира се, когато работим с наши запазени класове, няма смисъл да използваме статичните методи, защото идеята е да се възползваме от вече дефинирания шаблон.

Освен многократното използване на полезни регулярни изрази, предварителното компилиране има още едно предимство – повишава се скоростта на работа, защото се спестява времето за компилация на израза – той е вече компилиран във включеното асембли.

Кога да използваме регулярни изрази

Регулярните изрази представляват значително удобство при текстообработката. След като придобием опит в използването им, е лесно да се изкушим и да започнем да ги употребяваме при всяка възможна задача. Трябва обаче да си даваме сметка, че регулярните изрази крият някои неудобства.

Регулярните изрази със сигурност се поддържат доста трудно. Макар и разбираем, техният синтаксис не е лесен за четене. В момента, в който ни трябват да ги пишем за първи път, вероятно ще имаме достатъчно време да им отделим внимание и да ги обмислим добре. Обаче в някой следващ момент, в който ще ни се наложи да ги използваме или да ги променяме, е много вероятно изразите да ни изглеждат неразбираеми, именно защото трудно се четат. Това важи в още по-голяма степен, ако се налага да променяме или дебъгваме написани от някой друг регулярни изрази. Истина е, че коментарите донякъде помагат за разбирането на шаблоните, но като цяло поддръжката на регулярни изрази е трудна и бавна.

Както и изобщо при писането на код, добре е да се придържаме към по-прости регулярни изрази. Разбира се тяхната сила за някое елегантно

решение на практически проблем обикновено се проявява при по-сложни и неочевидни конструкции, но дори случаят да е такъв, добре е да слагаме достатъчно коментари, за да може някой след нас да разбере какво правим и защо го правим с този вид на шаблона. Като допълнителна препоръка, хубаво е проблемите да се решават на части и с по-прости регулярни изрази. Така можем едновременно да се възползваме от тяхната мощ и да им осигурим по-лесна поддръжка.

Няколко думи за ефективността

Регулярните изрази понякога може да имат изключително ниска ефективност. Това също е една от причините да внимаваме при тяхната употреба. Проблемите се ефективността най-често произтичат именно от неправилно използване на регулярните изрази. Следният код например може да забави значително дори бърз компютър (на 1,7 GHz машина този код не успява да завърши за една нощ, а повече не е изчакван):

```
string text = "aaabacababababaccbacbcbccacbcbccbacccccc";
string pattern = @"(\w*ab|\w*ac|\w*aa|\w*c)*cccccc";
Match m = Regex.Match(text, pattern);
```

Причината за ниската ефективност, на която могат да са жертва регулярните изрази, е в начина им на изпълнение. За да поддържат изразителната мощ и възможностите, които описахме, регулярните изрази използват споменатия вече механизъм на недетерминирани крайни автомати. В горния пример символите за количество * карат машината да извършва мащабно търсене с връщане назад (backtracking) докато отхвърли всички неподходящи поднизове при търсенето. При това заради лакомите звездички търси първо най-големите съвпадения, което допълнително я забавя.

Търсенето с връщане назад е основният проблем при работата с шаблони. Доброто познаване на действието на машината на регулярните изрази и достатъчно опит са незаменими за избягване на неправилната употреба на шаблоните, която намалява ефективността и подкопава ползата от тях. Добре е да се опитваме да пишем прости регулярни изрази, за да можем да ги осмислим. Можем да си ги представяме като граф (какъвто е и самият краен автомат) – ако графът има много цикли, вероятно обработката с шаблона ще е бавна.

Пример за неподходящ backtracking

Ще покажем само един кратък пример, който да ни даде представа за какво трябва да внимаваме. В частта за "мързеливите" метасимволи показахме как трябва да се употреби конструкцията **<.+?>** вместо **<.+>**, за да уловим само отварящия таг, а не и затварящия.

Тази конструкция обаче е неефективна. Да разгледаме примерния текст **<body>text</body>** и да видим как действа машината тук. Първият символ "<" коректно отговаря на шаблона и търсенето продължава нататък. Тук

има символ "b". Той изпълнява условието на точката в шаблона и "мързеливото" търсене "казва" на машината, че е открила минимален брой съвпадения с точката и да продължава нататък. Тук в шаблона има >, но в текста има "o". Следователно не може да има успешно съвпадение и машината се връща назад, за да опита този път с два символа, които да удовлетворят **.+?**. Вижда се, че чак след четири връщания ще имаме резултат – това е пример за несъобразяване с търсенето с връщане назад.

По-добрият вариант е конструкцията **<[^>]+>**. При нея машината директно обхожда символите, докато стигне до затварящата скоба и понеже плюсът е "лаком", приема това за успешно съвпадение и не се връща назад.

Друг пример за лош израз е **.*(pattern).***, когато просто търсим съвпадение с **pattern**. Тук от ограждането с **.*** няма никакъв смисъл, а излишният backtracking забавя обработката на израза.

В заключение, трябва да внимаваме кога шаблоните ни може да попаднат в ситуация да забавят твърде много изпълнението на програмата. По-простите изрази и тук помагат за по-лесно избягване на този проблем и добра поддръжка.

Няколко регулярни израза от практиката

Ще покажем още няколко регулярни израза, които биха могли да се използват за практическа употреба при разработката на реален софтуер и адресират често срещани задачи. Към тях може да се добави и подробно разгледаният вече пример за извличане на HTML тагове от текст.

Размяна на първите две думи в низ

Чрез символите за количество повторения, този пример лесно може да бъде разширен и за по-общи случаи (разменяне на m-тата и n-тата дума и др.). Изразът премахва и празните символи в началото на текста.

```
string text = " няма бира, дай ракия!";
string pattern = @"^\s*(\w+)(\W+)(\w+)";
string newText = Regex.Replace(text, pattern,"$3$2$1");
Console.WriteLine(newText);
// Output: бира няма, дай ракия!
```

Парсване на декларации <var>=<value>

Подобен израз вече разгледахме в един от примерите. Предполага се, че разглеждаме файл, в който тези декларации са подредени по една на ред. текста. Втората част на шаблона използва **\S** вместо **\w** заради възможността от значи като "%" и др.

```
string text = "server=mail.bg\nuser=u1\npass=!prl17";
string pattern = @"^(\w+?)\s*=\s*(\S+)\s*$";
```

```
MatchCollection matches = Regex.Matches(text, pattern,
   RegexOptions.Multiline);
foreach (Match m in matches)
{
   Console.WriteLine("var={0} value={1}",
      m.Groups[1], m.Groups[2]);
}
/* Output:
* var=server value=mail.bg
* var=user value=u1
* var=pass value=!prl17
*/
```

Парсване на дати

Това е задача, с която вероятно се е сблъсквал всеки програмист. Показаният в този пример израз не е перфектен (не налага ограничения дните да са до 31, а месеците до 12, а и не отчита кои месеци колко дни позволяват), но върши сравнително добра работа. Напълно коректен за задачата израз се конструира доста тромаво – ограничаването на числа в някакви интервали по принцип е задача, която не е съвсем свойствена за регулярните изрази. Тяхната сила е в изискванията към формата на текстовата информация.

```
string text = "17.03.2004 12:11:05";
string pattern =
   @"\A(?<day>\d{1,2})      # day in the beginning
   (\.|\/)                  # separator (. or /)
   (?<month>\d{1,2})        # month
   (\.|\/)                  # separator (. or /)
   (?<year>(19|20)?\d{2})   # year (19XX, 20XX or XX)
   \s+                      # whitespace
   (?<hour>\d{1,2})         # hour
   :                        # separator
   (?<min>\d{1,2})          # minutes
   (:(?<sec>\d{1,2}))?      # seconds (optional)";

Match match = Regex.Match(text, pattern,
   RegexOptions.IgnorePatternWhitespace);
if (match.Success)
{
   GroupCollection gr = match.Groups;
   Console.WriteLine("day={0} month={1} year={2}\n" +
      "hour={3} min={4} sec={5}",
      gr["day"], gr["month"], gr["year"],
      gr["hour"], gr["min"], gr["sec"]);
}
else
{
```

```
  Console.WriteLine("Invalid date and time!");
}
/* Output:
 * day=17 month=03 year=2004
 * hour=12 min=11 sec=05
 */
```

Премахване на път от името на файл

Шаблонът покрива както UNIX, така и Windows стил на изписване на пътя до файла. Лакомата звездичка позволява да се стигне до последната наклонена черта преди самото име на файла.

```
string fileName = @"/home/nakov/sample.tar.gz";
string fileOnly = Regex.Replace(fileName, @"^.*(\\|/)", "");
Console.WriteLine(fileOnly);
// Output: sample.tar.gz
```

Валидация на IP адреси

Тук можем да видим още един пример за това как трябва да преценяваме за себе си дали искаме абсолютна коректност или не дотам коректна проверка е достатъчна, за да осигури сигурността на програмата. Първият израз улавя правилно всички IP адреси и поставя условия за формат, но не ограничава числата до 255. Вторият пример прави това, но се вижда колко по-тромав става изразът. Използваме non-capturing groups, за да не бавим машината при търсенето с връщане.

```
string shortPattern = @"^(?:\d{1,3}\.){3}\d{1,3}$";
string longPattern = @"(?:(?:25[0-5]|2[0-4][0-9]|"
  + @"[01]?[0-9][0-9]?)\.){3}"
  + @"(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)";
```

Полезни Интернет ресурси

Има разбира се безбройно много други готови написани изрази. За почти всеки по-общ практически проблем може да се намери вече обмислен и изпробван израз в Интернет. Съществуват цели библиотеки от регулярни изрази, които са на разположение на всеки, решил да ги използва.

Добър пример в това отношение е сайтът http://www.regexlib.com/. Там може да се намерят редица полезни и интересни шаблони, които да улеснят разработката на вашата програма. С търсене в Интернет могат да се открият и други подобни библиотеки. Все пак добре е тези изрази да не се използват съвсем на готово. Повечето са въвеждани от произволни потребители и не винаги са точни и проверени във всички ситуации. Добра практика е шаблоните винаги да се преглеждат внимателно, преди да ги използваме.

За разработчиците, които предпочитат сами да пишат регулярните си изрази, Интернет предлага и много сайтове с обяснения за техния синтаксис и принцип на действие. Особено добър в това отношение е сайтът http://www.regular-expressions.info/, който е задължителен за всеки, решил да се занимава по-сериозно с регулярни изрази. Сайтът предлага примери, обяснения, таблици и изобщо всичко необходимо.

В Интернет могат да бъдат открити множество полезни програми за трениране с регулярни изрази и проверки за тяхната коректност. Такива са например **Regex Buddy** (http://www.regexbuddy.com/), която е платена, но предлага отлични възможности и интуитивен интерфейс; **Regex Coach** (http://www.weitz.de/regex-coach/), който има версия и за Linux, и **The Regulator** (http://regex.osherove.com/). Други подобни програми могат да бъдат намерени в категорията Regular Expressions в SharpToolbox (http://www.sharptoolbox.com/Pages/Category00d1e9e0-5976-48e1-a19d-304c76303239.aspx).

Разбира се, информация относно регулярните изрази и по-пълно описание на техния синтаксис в .NET, както и примери за употребата им, може да се намери в MSDN Library.

Инструментът The Regulator

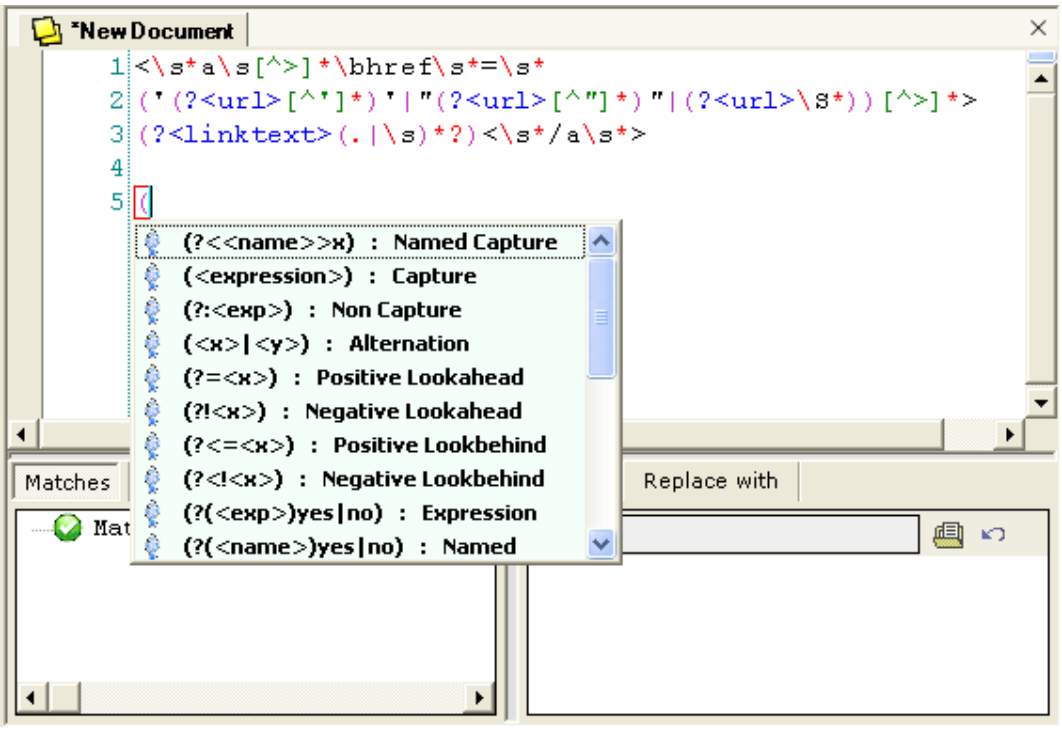
На края на темата ще демонстрираме една от гореизброените програми, инструментът The Regulator. Тя е с достатъчно лесен и интуитивен интерфейс, а същевременно предлага няколко интересни възможности, като генериране на .NET код и асемблита, търсене в RegexLib и др. Може да бъде изтеглена и инсталирана от адреса, който вече дадохме по-горе. Нека покажем накратко възможностите на програмата.

Стартираме програмата и в прозореца **New Document** въвеждаме нашия регулярен израз. Ще използваме за пример израза:

```
<\s*a\s[^>]*\bhref\s*=\s*('(?<url>[^']*)'|"(?<url>[^"]*)"|(?<url>
\S*))[^>]*>(?<linktext>(.|\s)*?)<\s*/a\s*>
```

за извличане на хипервръзки от HTML документ.

При въвеждането можем да обърнем внимание на някои от екстрите, които редакторът предлага. Например при селектиране на някои отваряща или затваряща скоба, тя се оцветява в червен правоъгълник, заедно със съответната ѝ. При изписването на отваряща скоба ни се показва и списък с auto-complete възможности за продължаване на израза. Ако ни потрябва някой метасимвол, можем лесно да го достъпим и чрез десен бутон и опцията **Quick Add**.

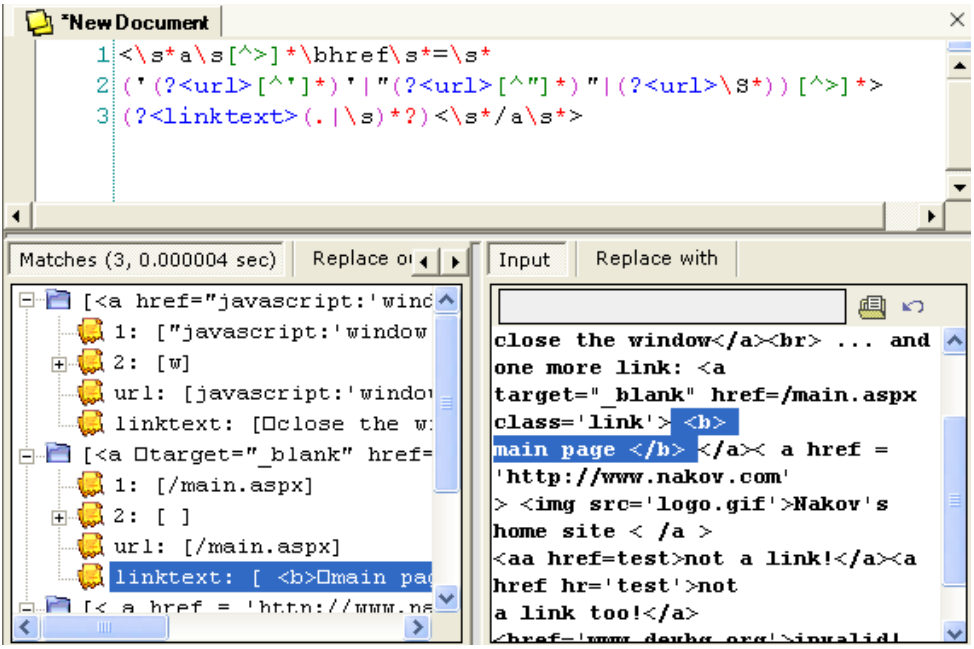


Нека сега въведем следния текст в прозореца **Input**:

```
<html> This is a hyperlink:
<a href="javascript:'window.close()'">
close the window</a><br> ... and one more link: <a
target="_blank" href=/main.aspx class='link'> <b>
main page </b> </a>< a href = 'http://www.nakov.com'
> <img src='logo.gif'>Nakov's home site < /a >
<aa href=test>not a link!</a><a href hr='test'>not
a link too!</a><href='www.devbg.org'>invalid!</a>
</html>
```

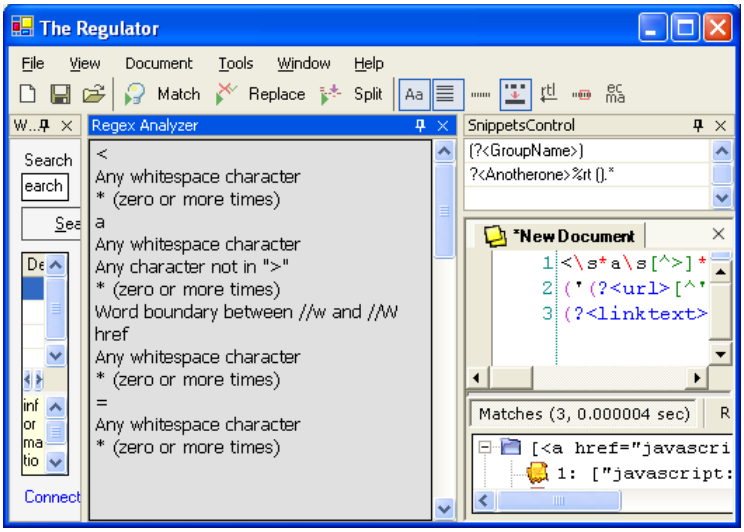
и да натиснем бутона **Match** на лентата с инструменти. В прозореца **Matches** долу вляво получаваме списък със съвпаденията, открити с нашия шаблон във въведения текст.

Виждаме, че съвпаденията се показват в дървовиден вид – на най-високо ниво е самото съвпадение с целия израз, а под него следват групите в израза и съвпаденията с тях. Когато избираме някоя група вляво, вдясно селекцията автоматично се премества на съответното място.



По подобен начин можем да работим с функциите **Replace** и **Split**. В прозореца **Matches** можем да превключим на **Splits** и с натискането на бутона **Split** да видим как се разделя текста от **Input** по нашия шаблон. Аналогично, ако въведем някакъв шаблон за заместване (replacement pattern) в прозореца **Replace With**, който е при **Input**, и натиснем бутона **Replace**, то в прозореца **Matches** превключваме на **Replace Output** и можем да видим какъв би бил резултатът от заместването.

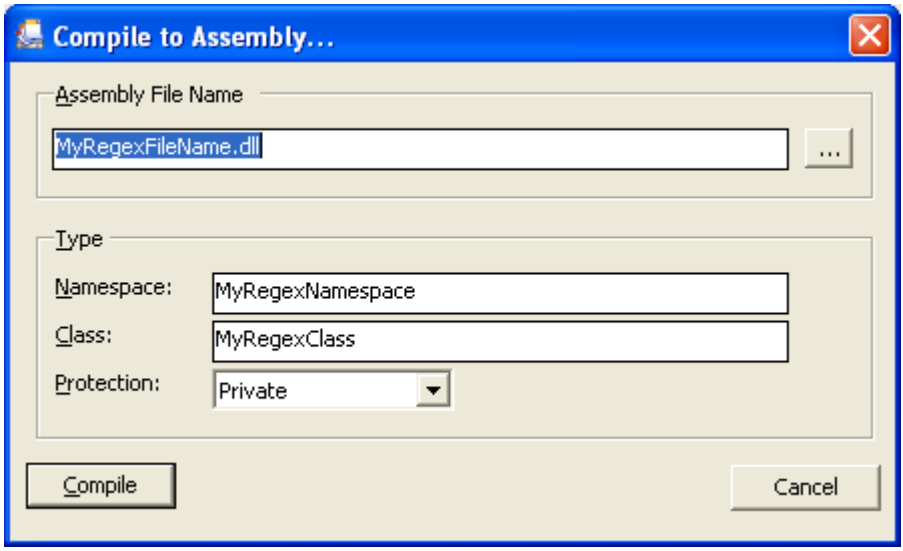
Нека сега натиснем **Ctrl+Shift+A**. В прозореца **RegexAnalyzer** се появява подробно описание на това какви съвпадения намира нашия израз, включително се вижда как са обособени групите и какви съвпадения в частност намират те:



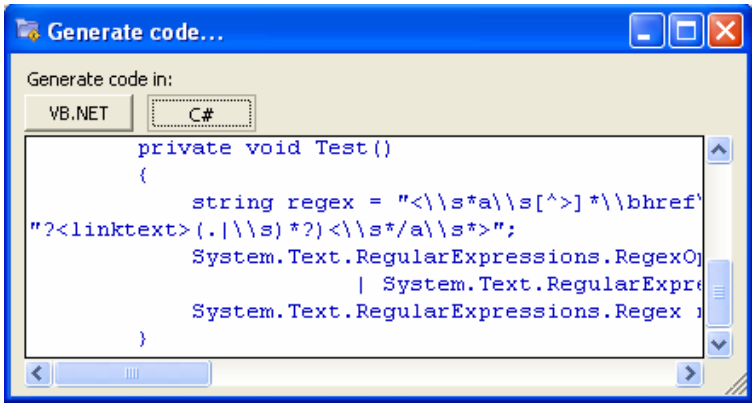
Прозорецът **SnippetsControl** ни позволява да запазваме там често използвани части от регулярни изрази (с двойно щракване върху празния ред там въвеждаме нов snippet) и после да ги въвеждаме лесно в нашия шаблон (отново с двойно щракване върху искания snippet).

В прозореца **WebSearch** можем да търсим при налична връзка с Интернет директно в библиотеката RegexLib по ключови думи за регулярни изрази, които ни интересуват. Получаваме списък с резултати, за всеки от които има описание и примери.

Менюто **Tools** предлага няколко интересни възможности, които споменахме. Там можем да активираме форма за изпращане на наш регулярен израз в RegexLib. Можем също да генерираме .NET асембли с израза, което да запазим като DLL библиотека, чрез опцията **Compile to Assembly**.



С натискането на **Ctrl+K** активираме опцията **Generate Code**, която ни предоставя готов .NET код за използване на нашия израз. Формата поддържа C# и VB.NET:



Упражнения

1. Опишете накратко какво представляват регулярните изрази. Кои са основните елементи на езика на регулярните изрази? Какви метасимволи познавате?
2. Опишете накратко средствата на .NET Framework за работа с регулярни изрази - основните класове и по-важните им методи.
3. Напишете програма, която с помощтта на регулярен израз по дадена последователност от символи (цел) и даден текст извлича от текста всички думи, които съдържат зададената цел в себе си като подниз.
4. Възможно ли е чрез регулярен израз да се провери дали скобите в даден числов израз са поставени правилно (дали за всяка отваряща скоба има съответстваща затваряща). Защо?
5. Напишете програма, която с помощта на регулярен израз валидира реални числа във формат: цяла, следвана от дробна част. Например числата "0", "33", "-2381.78132", "4.3347", "12.00" и "0.34" се считат за валидни, а числата "+3", "—2", "24 543", "01.23", "12. ", "11,23", "12e7" – за невалидни.
6. Напишете програма, която изважда от даден текстов документ всички поднизове, които приличат на e-mail адрес (последователности от символи във формат <identifier>@<host>... <domain>). Използвайте подходящ регулярен израз.
7. Напишете програма, която изважда от даден текстов документ всички низове, които приличат на URL адреси (поднизове във формат www.<host>...<domain> и поднизове, започващи с "http:// ").
8. Напишете програма, която с помощта на регулярен израз по даден URL адрес във формат [protocol]://[server]/[resource] извлича от него отделните му елементи – [protocol], [server] и [resource]. Например за URL http://www.devbg.org/forum/index.php трябва да извлече [protocol] = "http", [server] = "www.devbg.org" и [resource] = "/forum/index.php".
9. Даден е речник с думи, който представлява текст във формат "дума значение", по една речникова единица на всеки ред (значението може да се състои от няколко думи). Да се състави програма, която по дадена дума намира значението й в речника. Използвайте регулярни изрази и групи за парване на текста.
10. Напишете програма, която претърсва даден текст за дадена дума и намира и отпечатва всички изречения, в които тази дума се среща. Можете да считате, че всяко срещане на някой от символите ".", "!" и "?" означава край на изречение. Например в текста "\tНалей ми бира! Истина бирата заради тези регулярни изрази. Ще сложа две-три в камерата.\n \t Отивам до магазина за още бира." думата "бира" се среща само в първото и последното изречение. За разделяне на

изреченията едно от друго използвайте регулярни изрази. Подходящ ли е изразът `\s*(.|\s)*(\.|\!|\?)` и защо?

11. Напишете програма, която извлича от даден текст всички цели числа без знак и ги записва в масив от символни низове. За целта използвайте метода **`Regex.Split`**.

12. Напишете програма, която заменя в даден HTML документ всички хипервръзки ... с метаописание на тези връзки във формат [url href=...]...[/url]. Програмата трябва да се справя с вложени тагове и дори с вложени хипервръзки (въпреки че това не е позволено в езика HTML). Използвайте регулярни изрази и метода **`Regex.Replace`**.

13. Напишете програма, която обръща думите в дадено изречение в обратен ред. Например изречението "Брала мома къпини." трябва да се преобразува в "Къпини мома брала.". Използвайте метода **`Regex.Replace`** заедно с **`MatchEvaluator`**.

Използвана литература

1. Светлин Наков, Регулярни изрази – http://www.nakov.com/dotnet/lectures/Lecture-9-Regular-Expressions-v1.0.ppt
2. Regular Expression Tutorial – http://www.regular-expressions.info/tutorial.html
3. Brad Merrill, C# Regular Expressions – http://windows.oreilly.com/news/csharp_0101.html
4. Yashavant Kanetkar, The Regular Expressions – http://www.funducode.com/csharpart/csarticle30.htm
5. Building a Regular Expression Library Source Listing – http://haacked.com/articles/1465.aspx
6. MSDN library – http://msdn.microsoft.com/library/

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 11. Вход и изход

Необходими знания

- Базови познания за общата система от типове в .NET (Common Type System)
- Базови познания за езика C#
- Базови познания за управление на паметта и ресурсите в .NET Framework
- Базови познания по файлови системи

Съдържание

- Какво представляват потоците?
- Потоците в .NET Framework. Базови и преходни потоци
- Типът **`System.IO.Stream`**. Основни операции
- Буферирани потоци
- Файлови потоци
- Четци и писачи. Двоични и текстови четци и писачи
- Операции с файлове. Класове **`File`** и **`FileInfo`**
- Работа с директории. Класове **`Directory`** и **`DirectoryInfo`**
- Наблюдение на файловата система с **`FileSystemWatcher`**
- Работа с **`IsolatedStorage`**

В тази тема ...

В настоящата тема ще разгледаме начина, по който се осъществяват вход и изход от дадена програма в .NET Framework. Ще представим различните видове потоци – абстракцията, която позволява връзката на програмата с някакво устройство за съхранение на данни. Ще обясним работата на четците и писачите, които обвиват потоците и така улесняват работата с тях. Накрая, ще прегледаме какви средства предоставя .NET Framework за работа с файлове и директории и за наблюдение на файловата система.

Какво представляват потоците?

Потоците в обектно-ориентираното програмиране са една абстракция, с която се осъществява вход и изход от дадена програма. Потоците в C# като концепция са аналогични на потоците в други обектно-ориентирани езици, напр. Java, C++ и Delphi (Object Pascal).

Потокът е подредена серия от байтове, която служи като абстрактен канал за данни. Този виртуален канал свързва програмата с устройство за съхранение или пренос на данни (напр. файл върху хард диск), като достъпът до канала е последователен. Потоците предоставят средства за четене и запис на поредици от байтове от и към устройството. Това е стандартният механизъм за извършване на входно-изходни операции в .NET Framework.

Потоци – пример

Следната програма реализира копиране на файлове чрез потоци. Тя създава копие на Notepad (**notepad.exe**), стандартния текстов редактор на Windows.

BinaryFileCopier.cs

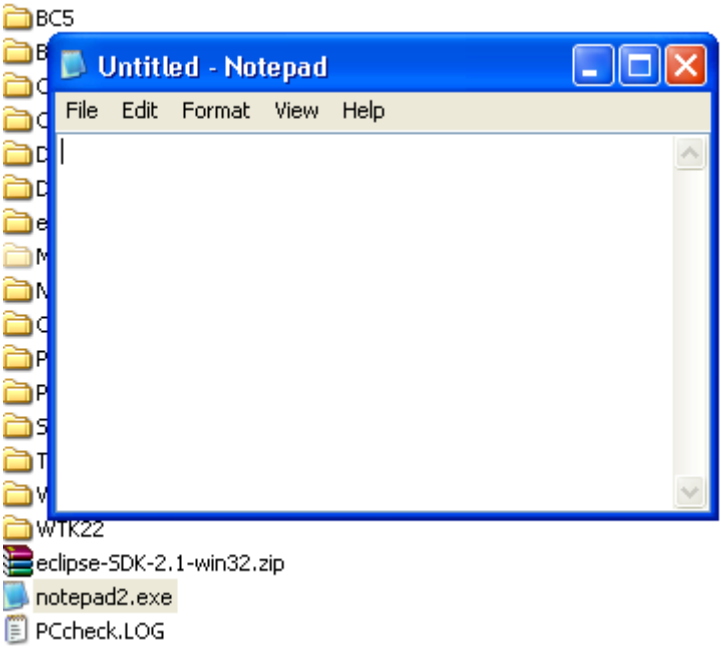
```
using System;
using System.IO;

public class BinaryFileCopier
{
   public const string INPUT_FILE = @"C:\Windows\notepad.exe";
   public const string OUTPUT_FILE = @"C:\notepad2.exe";

   static void Main()
   {
      using (
         FileStream inFile = new FileStream(INPUT_FILE,
            FileMode.Open),
         outFile = new FileStream(OUTPUT_FILE, FileMode.Create))
      {
         byte[] buf = new byte[1024];
         while (true)
         {
            int bytesRead = inFile.Read(buf, 0, buf.Length);
            if (bytesRead == 0)
               break;
            outFile.Write(buf, 0, bytesRead);
         }
      }
   }
}
```

Как работи примерът?

В горната програма със създаването на **inFile** и **outFile** от клас **FileStream**, създаваме два двоични потока, като ги свързваме с два файла, намиращи се в съответните директории (съответно **notepad.exe** и **notepad2.exe**). Другите параметри в конструктора показват, че първият файл се отваря като вече съществуващ, докато вторият се създава при изпълнението на програмата.

Използваната **using** клауза гарантира затварянето на използваните в нея потоци след приключване на работа с тях. Следва цикъл, който чете байтове от **notepad.exe** (като използва метода **Read()** на **inFile** обекта), записва ги в междинния масив от байтове **buf**, след което записва съдържанието на **buf** във файла **notepad.exe**. **Read()** връща действително прочетените байтове – те може да бъдат и по-малко от заявените. Действително прочетените байтове се запиват в изходния файл. Когато **inFile.Read()** върне **0**, входният файл вече е прочетен и копирането приключва.

Може да се провери, че изпълнението на програмата води до създаване на файл **notepad2.exe**, копие на традиционния **notepad.exe** от Windows директорията, който има същата функционалност:



Потоците в .NET Framework

Потоците в .NET Framework се делят на две групи – **базови** и **преходни**. И едните, и другите, наследяват абстрактния клас **System.IO.Stream**, базов за всички потоци.

Базови потоци (base streams)

Базовите потоци пишат и четат директно от някакъв външен механизъм за съхранение, като файловата система (например класът **`FileStream`**), паметта (**`MemoryStream`**) или данни, достъпни по мрежата (**`NetworkStream`**). По-нататък ще разгледаме класа **`FileStream`** в точката "Файлови потоци".

Преходни потоци (pass-through streams)

Преходните потоци пишат и четат от други потоци (най-често в базови потоци), като при това посредничество добавят допълнителна функционалност, например буфериране (**`BufferedStream`**) или кодиране (**`CryptoStream`**). По-подробно ще разгледаме **`BufferedStream`** в точката "Буферирани потоци".

Основни операции с потоци

Когато работим с потоци, върху тях можем да извършваме следните основни операции:

- **Конструиране** (създаване) – свързваме потока с механизма за пренос (съхранение) на данните (в случай на базов поток) или с друг поток (в случай на преходен поток). При това конструиране подаваме необходимата информация. Например, в случай на файлов поток подаваме име на файл и режим, в който го отваряме. В примера за копиране на файл вече показахме конструиране на файлов поток.
- **Четене** – извличане на данни от потока по специфичен за него начин. Извличането се извършва последователно, започвайки от текущата позиция.
- **Запис** – изпращат се данни в потока по специфичен за него начин. Записът става от текущата позиция.
- **Позициониране** – премества текущата позиция на потока (ако потокът поддържа позициониране). Можем да позиционираме спрямо текуща позиция, спрямо начало на потока или спрямо края на потока. Някои потоци не поддържат позициониране (напр. **`NetworkStream`**).
- **Затваряне** – приключваме работата с потока и освобождаваме ресурсите, свързани с него. Например, ако потокът е файлов, записваме на диска данните от вътрешните буфери, които не са още записани и затваряме файла.
- Други операции – изпразване на вътрешните буфери (flush), асинхронно четене/запис (вж. темата "Многонишково програмиране и синхронизация") и други.

Типът System.IO.Stream

Абстрактният клас **System.IO.Stream** е базов за всички потоци в .NET Framework. Той се наследява от файловите, мрежовите и всички останали видове потоци. В него са дефинирани методи за извършване на основните операции, описани по-горе.

Не всички потоци поддържат четене, запис и позициониране. Кои от тези операции се поддържат, може да се провери със свойствата **CanRead**, **CanWrite** и **CanSeek**. Ако потокът поддържа позициониране, дефинирани са и свойствата **Position** и **Length**.

Класът има и поле **Stream.Null** (също от клас **Stream**), което игнорира всички опити за четене и запис и може да бъде използвано в някои специфични ситуации, например за пренасочване на ненужен изход, без да се хабят излишно системни ресурси.

Четене от поток

За четене на данни от поток се използва методът **int Read(byte[] buffer, int offset, int count)**. Той чете най-много **count** на брой байта от текущата позиция на входния поток, увеличава позицията и връща броя прочетени байтове или **0** при достигне края на потока.

Четенето може да блокира за неопределено време. Например, ако при четене от мрежа извикаме метода **NetworkStream.Read(…)**, а не са налични данни за четене, операцията блокира до тяхното получаване. В такива случаи е уместно да се използва свойството **NetworkStream.DataAvailable**, което показва дали в потока има пристигнали данни, които още не са прочетени, т. е. дали последваща операция **Read()** ще блокира или ще върне резултат веднага.

Неправилна употреба на DataAvailable

Имайте предвид, че **DataAvailable** ще върне **false** дори ако данни са изпратени от отсрещната страна, но те все още не са пристигнали в потока. Дали е стигнат края на потока можем да разберем единствено като извикваме метода **Read()** и той върне **0**. Затова, долният код е некоректен:

```
NetworkStream myNetworkStream = ...;
if (myNetworkStream.CanRead)
{
   byte[] myReadBuffer = new byte[1024];
   do
   {
      int numberOfBytesRead = myNetworkStream.Read(
         myReadBuffer, 0, myReadBuffer.Length);
      // Do something with the data in myReadBuffer
   }
```

```
   while (myNetworkStream.DataAvailable);
   // DataAvailable==false does not mean "end of stream"
}
```

В общия случай горният примерен код няма да прочете всичко, а ще спре при първото забавяне в четенето.

`DataAvailable` може да стане `false` преди да е достигнат краят на потока. Съобразявайте се с това.

Неправилна употреба на Read()

При четене от поток с метода **`Read(…)`** или с методи, които разчитат на **`Read(…)`**, броят на прочетените байтове може да е по-малък от броя на заявените. Във връзка с това, ще дадем един пример за некоректно копиране на файлове. Следният код в общия случай работи неправилно:

```
using (
   FileStream inFile = new FileStream("input.bin",FileMode.Open),
   outFile = new FileStream("output.bin", FileMode.Create))
{
   byte[] buf = new byte[4096];
   while (true)
   {
      // Bad practice! Read() may not read buf.Length bytes!
      if (inFile.Read(buf, 0, buf.Length) == 0)
      {
         break;
      }
      outFile.Write(buf, 0, buf.Length);
   }
}
```

Както е посочено в коментара в примера, **`Read(…)`** не е задължително да прочете **`buf.Length`** байта. Тогава записът на **`buf.Length`** байта в **`outFile`** не би било коректно.

Четенето от поток може да прочете по-малко от заявения брой байтове, дори ако не е достигнат краят на потока. Съобразявайте се с това, за да не допускате грешки.

Правилното копиране вече бе демонстрирано в началото на темата, при копирането на **`notepad.exe`**. Оставяме на читателя да направи сравнение.

Писане в поток

Методът **`Write(byte[] buffer, int offset, int count)`** записва в изходния поток **`count`** байта, като започва от зададеното отместване в байтовия

масив. И тази операция е блокираща, т.е. може да предизвика забавяне за неопределено време. Не е гарантирано, че байтовете, записани в потока с **Write(…)**, са достигнали до местоназначението си след успешното изпълнение на метода. Възможно е потокът да буферира данните и да не ги изпраща веднага.

Изчистване на работните буфери

Методът **Flush()** изчиства вътрешните буфери, като изпраща съдържащите се в тях данни към механизма за съхранение (пренос). След успешното приключване на изпълнението на **Flush()** е гарантирано, че всички данни, записани в потока, са изпратени към местоназначението си, но няма гаранция, че ще пристигнат успешно до него.

Ако при писане в поток не се извиква Flush(), няма гаранция, че данните, записани в потока, са изпратени.

Затваряне на поток

Методът **Close()** извиква **Flush()**, затваря връзката към механизма за съхранение (пренос) на данни и освобождава използваните ресурси.

Алтернатива на **Close()** е **using** конструкцията. Когато използваме **using**, дефинираме програмния блок, в който е видим създавания обект. При достигане края на блока, гарантирано се извиква методът **Dispose()** на посочения в клаузата обект, а той вътрешно извиква **Close()**.

Винаги затваряйте потоците, които използвате, за да не предизвиквате загуба на ресурси!

Правилно затваряне на поток

Типично за начинаещия програмист е да напише следния код за работа с поток:

```
Stream stream = ...; // Obtain opened stream
// Do something with the stream here
stream.Close();
```

Проблемът на този код е, че ако по време на работата с отворения поток възникне изключение, операцията **Close()** няма да се изпълни и потокът ще остане отворен.

Това е сериозен проблем, защото води до потенциална загуба на ресурси, а ресурсите са ограничени и не трябва да се пропиляват. При изчерпване на ресурсите приложението започва да става нестабилно и да предизвиква неочаквани грешки и сривове.

Правилната работа с потоци изисква затварянето им да бъде гарантирано след приключване на работата с тях или чрез **using** конструкцията в C# или чрез употребата на **try-finally** блок.

Ето как можем да използваме конструкцията **using** за правилно освобождаване на поток:

```
Stream stream = ...; // Obtain opened stream
using (stream)
{
   // Do something with the stream here
}
// The stream will be automatically closed here
```

Ето и алтернативният вариант в **try-finally** конструкция:

```
Stream stream = ...; // Obtain opened stream
try
{
   // Do something with the stream here
}
finally
{
   // Manually close the stream after finishing working with it
   stream.Close();
}
```

И в двата варианта е предвиден случаят, в който по време на работа възниква изключение. В този случай потокът ще бъде затворен преди да бъде обработено изключението.

Промяна на текущата позиция в поток

Методът **Seek(int offset, SeekOrigin origin)** премества текущата позиция на потока с **offset** на брой байта спрямо зададена отправна точка (начало, край или текуща позиция на потока). Методът е приложим за потоците, за които **CanSeek** връща **true**, за останалите хвърля изключение **NotSupportedException**.

Методът **SetLength(long length)** променя дължината на потока (ако това се поддържа). Промяната на дължината на поток е рядко използвана операция и се поддържа само от някои потоци, например **MemoryStream**.

Буферирани потоци

Буферираните потоци използват вътрешен буфер за четене и запис на данни, с което значително подобряват производителността.

Когато четем данни от някакво устройство, при заявка дори само за един байт, в буфера попадат и следващите го байтове до неговото запълване.

При следващо четене, данните се взимат директно от буфера, което е много по-бързо. По този начин се извършва кеширане на данните, след което се четат кеширани данни.

При запис, всички данни попадат първоначално в буфера. Когато буферът се препълни или когато програмистът извика **`Flush()`**, те се записват върху механизма за съхранение (пренос) на данни.

Класът, който реализира буфериран поток в .NET Framework, е **`System.IO.BufferedStream`**. Този клас или негов наследник трябва да бъде използван, когато трябва да се подобри производителността на входно-изходните операции в приложението.

Файлови потоци

Файловите потоци в .NET Framework са реализирани в класа **`FileStream`**, който вече беше използван в примера за потоци. Като наследник на **`Stream`**, той поддържа всичките му методи и свойства (четене, писане, позициониране) и добавя някои допълнителни.

Създаване на файлове поток

В .NET Framework файлов поток се създава по следния начин:

```
FileStream fs = new FileStream(string fileName,
  FileMode [, FileAccess [, FileShare]]);
```

При конструирането, посочваме името на файла, с който свързваме потока (**`fileName`**), начина на отваряне на файла (**`FileMode`**), правата, с които го отваряме (**`FileAccess`**) и правата, които притежават другите потребители, докато ние държим файла отворен (**`FileShare`**).

`FileMode` може да има една от следните стойности:

- **`Open`** - отваря съществуващ файл.
- **`Append`** - отваря съществуващ файл и придвижва позицията веднага след края му.
- **`Create`** – създава нов файл. Ако файлът вече съществува, той се презаписва и старото му съдържание с губи.
- **`CreateNew`** – аналогично на **`Create`**, но ако файлът съществува, се хвърля изключение.
- **`OpenOrCreate`** – отваря файла, ако съществува, в противен случай го създава.
- **`Truncate`** – отваря съществуващ файл и изчиства съдържанието му, като прави дължината му `0` байта.

FileAccess и **FileShare** могат да приемат стойности **Read**, **Write** и **ReadWrite**. **FileShare** може да бъде и **None**.

Четене и писане във файлов поток

Четенето и писането във файлови потоци, както и другите по-рядко използвани операции, се извършват както при всички наследници на класа **Stream** – с методите **Read()**, **Write()** и т. н.

Файловите потоци поддържат пряк достъп до определена позиция от файла чрез метода **Seek(…)**.

Пример – замяна на стойност в двоичен файл

Ще дадем следния пример за работа с файлови потоци, който заменя дадена стойност в двоичен файл с друга:

Replacer.cs

```
using System;
using System.IO;

class Replacer
{
   const int BUFFSIZE = 16384;
   const byte SPACE_SYMBOL_CODE = 32;

   static void Main()
   {
      FileStream fs = new FileStream("file.bin",
         FileMode.Open, FileAccess.ReadWrite, FileShare.None);
      using (fs)
      {
         byte[] buf = new byte[BUFFSIZE];
         while (true)
         {
            int bytesRead =
               fs.Read(buf, 0, buf.Length);
            if (bytesRead == 0)
               break;
            for (int i=0; i<bytesRead; i++)
            {
               if (buf[i] == SPACE_SYMBOL_CODE)
                  buf[i] = (byte) '-';
            }
            fs.Seek(-bytesRead, SeekOrigin.Current);
            fs.Write(buf, 0, bytesRead);
         }
      }
   }
```

```
}
```

Как работи примерът?

В примера "**file.bin**" е бинарен файл от директорията **bin\Debug** на проекта. Преди изпълнението на програмата, в него можем да сложим произволно двоично или текстово съдържание, например горния сорс код. Примерът търси всички срещания на стойността **32** (ASCII кода на символа интервал) и я заменя с **45** (ASCII кода на символа тире). След като масивът от байтове **buf** се запълни с байтове от файла, правим замяната, след което връщаме позицията на потока и записваме коригираната информация върху старата. Получаваме бинарен файл, в който интервалите са заменени с тирета. Тъй като в случая "**file.bin**" съдържа текстова информация, той може да бъде отворен и разгледан с Notepad, както преди замяната, така и след нея. В общия случай, бинарни файлове могат да се разглеждат и във VS.NET чрез влачене и пускане.

Четци и писачи

Четците и писачите (readers and writers) в .NET Framework са класове, които улесняват работата с потоците. При работа например само с файлов поток, програмистът може да чете и записва единствено байтове. Когато този поток се обвие в четец или писач, вече са позволени четенето и записа на различни структури от данни, например примитивни типове, текстова информация и други типове. Четците и писачите биват двоични и текстови.

Двоични четци и писачи

Двоичните четци и писачи осигуряват четене и запис на примитивни типове данни в двоичен вид – **ReadChar()**, **ReadChars()**, **ReadInt32()**, **ReadDouble()** и др. за четене и съответно **Write(char)**, **Write(char[])**, **Write(Int32)**, **Write(double)** – за запис. Може да се чете и записва и **string**, като той се представя във вид на масив от символи и префиксно се записва дължината му – **ReadString()**, респ. **Write(string)**.

Двоични четци и писачи – пример

Следният пример демонстрира работата с двоични четци и писачи:

BinaryFilesDemo.cs

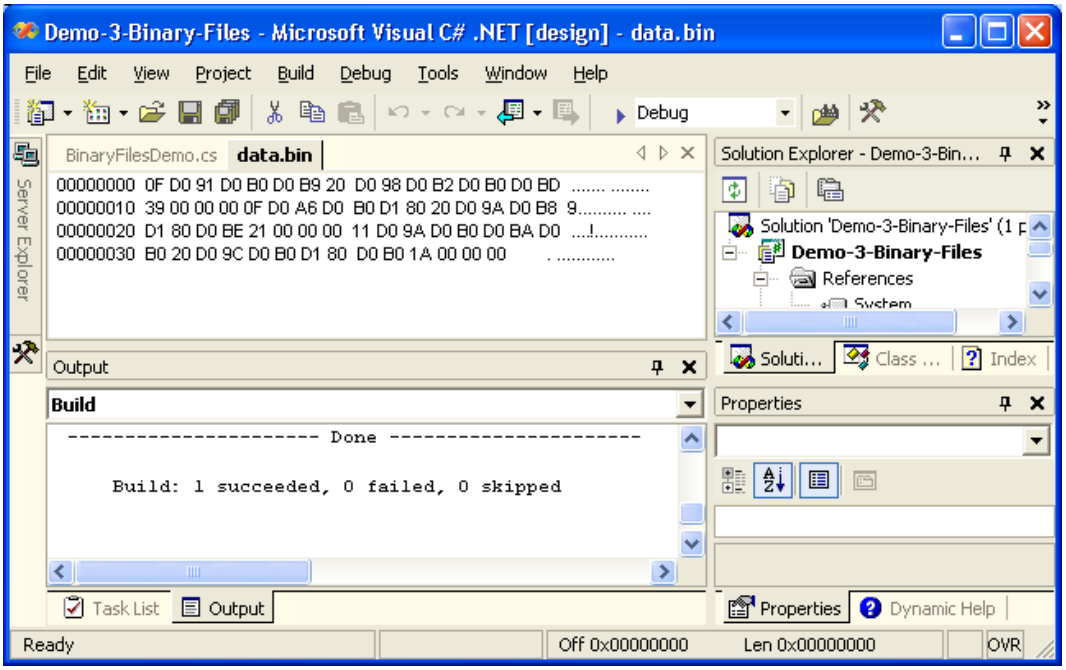
```
using System;
using System.IO;

class BinaryFilesDemo
{
  static void Main()
```

```
    {
        // Create the file
        FileStream fs = new FileStream("data.bin",FileMode.Create);
        using (fs)
        {
            /* Open a binary writer on the existing file stream and
            write some data */
            using (BinaryWriter writer = new BinaryWriter(fs))
            {
                AppendPerson(writer, "Бай Иван", 57);
                AppendPerson(writer, "Цар Киро", 33);
                AppendPerson(writer, "Кака Мара", 26);
            }
        }

        // Open the existing file
        fs = new FileStream("data.bin", FileMode.Open);
        using (fs)
        {
            //read data using binary reader
            using (BinaryReader reader = new BinaryReader(fs))
            {
                while (fs.Position < fs.Length-1)
                {
                    string name;
                    int age;
                    ReadPerson(reader, out name, out age);
                    Console.WriteLine("{0} - {1}", name, age);
                }
            }
        }
    }

    static void AppendPerson(BinaryWriter aWriter,
        string aName, int aAge)
    {
        aWriter.Write(aName);
        aWriter.Write(aAge);
    }

    static void ReadPerson(BinaryReader aReader,
        out string aName, out int aAge)
    {
        aName = aReader.ReadString();
        aAge = aReader.ReadInt32();
    }
}
```

Как работи примерът?

Примерът показва отварянето първо на двоичен писач, а после и на двоичен четец върху файловия поток **fs**, който пък на свой ред се отваря върху бинарния файл **"data.bin"**. Първо записваме във файла три записа с по две полета, а после ги прочитаме от файла и ги извеждаме на конзолата.

За добавянето и четенето на записи използваме двата статични метода **AppendPerson()** и **ReadPerson()**. Самото съдържание на файла може да се разгледа с вградения във VS.NET редактор за бинарни файлове (hex editor), най-лесно с влачене и пускане. Текстът в него не е четим, поради използваното кодиране UTF-8. Това е съдържанието на файла, разгледан с редактора за бинарни файлове:



Текстови четци и писачи

Текстовите четци и писачи осигуряват четене и запис на текстова информация, представена във вид на низове, разделени с нов ред. Базови текстови четци и писачи са абстрактните класове **TextReader** и **TextWriter**. Основните методи за четене и запис са следните:

- **ReadLine()** – прочита един ред текст.
- **ReadToEnd()** – прочита всичко от текущата позиция до края на потока.
- **Write(…)** – вмъква данни в потока на текущата позиция.

- **WriteLine(…)** – вмъква данни в потока на текущата позиция и добавя символ за нов ред.

Текстовите четци прочитат само текст, а писачите записват различни типове данни в текстов формат.

Всяка от изброените операции е блокираща операция. Това означава, че извикването на някои от описаните методи може да се забави известно време, например докато пристигнат или бъдат изпратени данните към техния източник.

Понеже текстовите четци и писачи работят с редове информация, следва да се отбележи, че символът за нов ред не е един и същ за всички платформи – за UNIX и Linux той е **LF** (**0x0A**), докато в Windows и DOS той е **CR+LF** (**0x0D + 0x0A**).

Тъй като класовете класове **TextReader** и **TextWriter** са абстрактни, за конкретни входно-изходни операции с текстови данни се използват техни наследници, например класовете:

- **StreamReader** – чете текстови данни от поток или файл.
- **StreamWriter** – записва текстови данни в поток или файл.
- **StringReader** – чете текстови данни от символен низ.
- **StringWriter** – записва текстови данни в символен низ.

Номериране на редове в текстов файл – пример

Следната програма отваря текстов файл и номерира редовете му, като използва класовете **StreamReader** и **StreamWriter**:

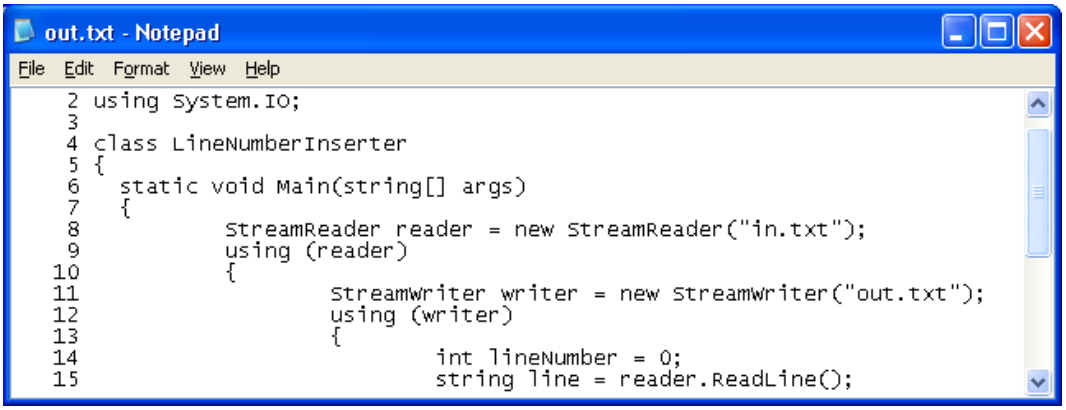
LineNumberInserter.cs

```
using System;
using System.IO;

class LineNumberInserter
{
  static void Main()
  {
    StreamReader reader = new StreamReader("in.txt");
    using (reader)
    {
      StreamWriter writer = new StreamWriter("out.txt");
      using (writer)
      {
        int lineNumber = 0;
        string line = reader.ReadLine();
        while (line != null)
        {
          lineNumber++;
```

```
            writer.WriteLine("{0,5} {1}", lineNumber, line);
            line = reader.ReadLine();
          }
        }
      }
    }
}
```

Как работи примерът?

След като свържем файловете **in.txt** и **out.txt** съответно с текстов четец и текстов писач, започва цикъл по редовете на входния файл, който копира всеки ред от **in.txt** в стринга **line** (**line = reader.ReadLine();**), след което записва **line** във файла **out.txt** (**writer.WriteLine(…)**). Записът на **line** се предшества от номера на реда, изведен в поле с ширина 5 символа. При достигане края на файла, **ReadLine()** връща **null**. Ето как изглежда резултатът от програмата, изпълнена върху нейния собствен сорс код:



Търсене на низ във файл – пример

Следната програма демонстрира търсене на дума (низ), използвайки **StreamReader**:

FindInFile.cs

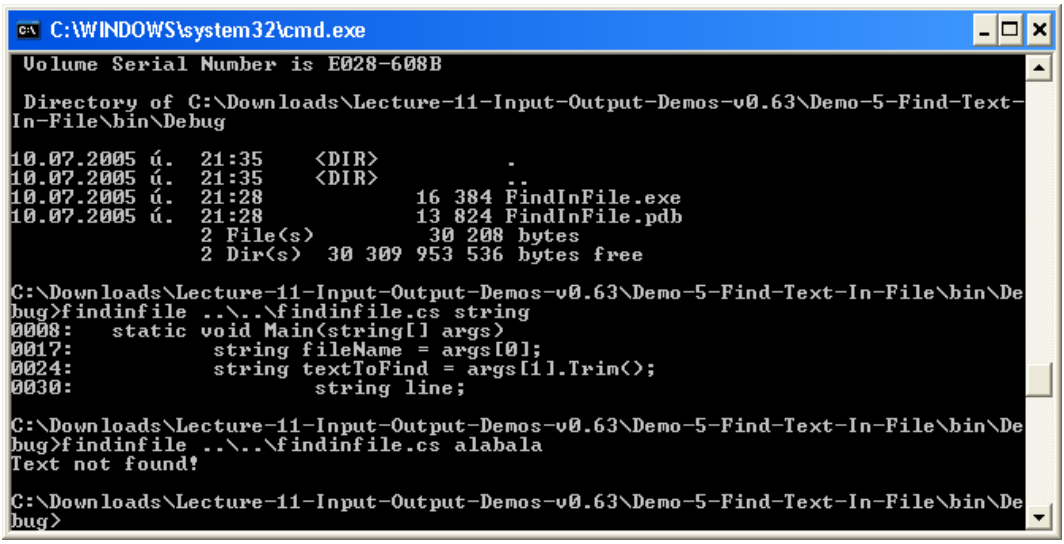
```
using System;
using System.IO;

class FindInFile
{
  static void Main(string[] args)
  {
    if (args.Length < 2)
    {
      Console.WriteLine(
```

```
            "Use: FindInFile file text");
         return;
      }

      string fileName = args[0];
      if (!File.Exists(fileName))
      {
         Console.WriteLine("Could not find file: " + fileName);
         return;
      }

      string textToFind = args[1].Trim();

      using (StreamReader reader = File.OpenText(fileName))
      {
         bool found = false;
         int lineNumber = 0;
         string line;
         while ((line = reader.ReadLine()) != null)
         {
            lineNumber++;
            if (line.IndexOf(textToFind) > -1)
            {
               Console.WriteLine("{0:0000}: {1}",
                  lineNumber, line);
               found = true;
            }
         }
         if (!found)
         {
            Console.WriteLine("Text not found!");
         }
      }
   }
}
```

Как работи примерът?

Програмата се стартира от командния ред. След като стартираме Command Prompt и отидем в директорията **bin\Debug** на проекта, стартираме **FindInFile.exe**. Ако не сме подали подходящи аргументи от командния ред, на конзолата се изписва указващото съобщение **"Use: FindInFile file text"**, т.е. като нулев аргумент подаваме файла, в който търсим, а като първи – търсения низ. Статичният метод **File.Exists(…)** проверява дали съществува файл, съответстващ на подадения низ, а **File.OpenText(…)** – отваря файла и установява позицията на потока в началото му. Класът **File** ще бъде разгледан съвсем скоро. Следващата част от кода обхожда редовете на отворения файл, търси стринга **textToFind** и евентуално, извежда реда, ако той

съдържа **textToFind**. Ето как изглежда изхода на програмата, когато търсим даден стринг в сорс кода на програмата – най-напред, когато търсения стринг се съдържа във файла, а след това – когато го няма.



Четци, писачи и кодирания

Тъй като на най-ниско ниво, всеки файл се състои от нули и единици, а четците и писачите работят с текстова информация, необходимо е кодиране, което да осъществява съответствието. В горните примери се използваше подразбращото кодиране UTF-8 и затова не споменахме изричното кодиране. Ако искаме да използваме друго кодиране, например windows-1251, начинът е следният:

```
Encoding win1251 = Encoding.GetEncoding("windows-1251");
StreamReader reader = new StreamReader("in.txt", win1251);
```

Задаването на кодиране windows-1251 е задължително при използване на текстови файлове на кирилица.

Ако искаме да създадем писач, а не четец, аналогията е пълна.

Потоци в паметта

Понякога се налага чрез средствата на потоците да четем данни от паметта или да записваме данни в паметта. Това се налага, когато някой метод, който искаме да използваме, приема като вход не масив от байтове (**byte[]**), а поток.

Четене от MemoryStream – пример

Със следващия пример ще демонстрираме четене от поток, който се съхранява в паметта:

```
using System;
using System.IO;
using System.Text;

class Test
{
   static void Main()
   {
      string name =
         "Национална академия по разработка на софтуер";
      byte[] data = Encoding.UTF8.GetBytes(name);

      MemoryStream ms = new MemoryStream(data);
      using (ms)
      {
         while (true)
         {
            int value = ms.ReadByte();
            if (value == -1)
            {
               break;
            }
            Console.Write("{0:X}", value);
         }
         Console.WriteLine();
      }
   }
}
```

Целта на примера е да отпечата даден символен низ като последователност от байтове в кодиране UTF-8, записани в шестнайсетичен вид. При изпълнение на примера се получава следният резултат:

```
D09DD0B0D186D0B8D0BED0BDD0B0D0BBD0BDD0B020D0B0D0BAD0B0D0B4D0B5D0
BCD0B8D18F20D0BFD0BE20D180D0B0D0B7D180D0B0D0B1D0BED182D0BAD0B020
D0BDD0B020D181D0BED184D182D183D0B5D180
```

Писане в MemoryStream – пример

Следващият пример илюстрира записване на данни в **MemoryStream** и извличането им като масив от байтове:

```
MemoryStream ms = new MemoryStream();
using (ms)
{
   ms.WriteByte(78);
   ms.WriteByte(97);
   ms.WriteByte(107);
   ms.WriteByte(111);
```

```
  ms.WriteByte(118);
}
byte[] data = ms.ToArray();
string s = Encoding.ASCII.GetString(data);
Console.WriteLine(s);
```

Примерът създава поток в паметта, записва в него последователно 5 байта и извлича записаните данни в **byte[]**, след което построява от тях символен низ. Резултатът от изпълнението на примера е:

```
Nakov
```

Операции с файлове. Класове File и FileInfo

Класовете **File** и **FileInfo** са помощни класове за работа с файлове. Те дават възможност за стандартни операции върху файлове като създаване, изтриване, копиране и др. В тях са дефинирани следните методи:

- **Create()**, **CreateText()** – създаване на файл.
- **Open()**, **OpenRead()**, **OpenWrite()**, **AppendText()** – отваряне на файл.
- **CopyTo(…)** – копиране на файл.
- **MoveTo(…)** – местене (преименуване) на файл.
- **Delete()** – изтриване на файл.
- **Exists(…)** – проверка за съществуване.
- **LastAccessTime** и **LastWriteTime** – момент на последен достъп и последен запис във файла.

В класа **File**, изброените методи са статични, а в класа **FileInfo** – достъпни чрез инстанция. Ако извършваме дадено действие еднократно (например създаваме един файл, след това го отваряме), класът **File** е за предпочитане. Работата с **FileInfo** и създаването на обект биха имали смисъл при многократното му използване. В примера за търсене на низ в текстов файл класът **File** вече бе използван.

File и FileInfo – пример

Да разгледаме и следния фрагмент от програма:

```
static void Main()
{
  StreamWriter writer = File.CreateText("test1.txt");
  using (writer)
  {
    writer.WriteLine("Налей ми бира!");
  }
```

```
    FileInfo fileInfo = new FileInfo("test1.txt");
    fileInfo.CopyTo("test2.txt", true);
    fileInfo.CopyTo("test3.txt", true);

    if (File.Exists("test4.txt"))
    {
        File.Delete("test4.txt");
    }

    File.Move("test3.txt", "test4.txt");
}
```

В примера, извикването на **`CreateText(…)`** създава файла **`test1.txt`** и отваря текстов писач върху него. С този писач можем да запишем някакъв произволен текст във файла. След създаването на **`FileInfo`** обект, копираме създадения файл в два други. Параметърът **`true`** означава, че при вече съществуващ файл **`test2.txt`**, респ. **`test3.txt`**, новият файл ще бъде записан върху стария. След това се проверява дали съществува файл **`test4.txt`** и се изтрива, след което **`test3.txt`** се преименува като **`test4.txt`**.

Работа с директории. Класове Directory и DirectoryInfo

Класовете **`Directory`** и **`DirectoryInfo`** са помощни класове за работа с директории. Ще изброим основните им методи, като отбележим, че за **`Directory`** те са статични, а за **`DirectoryInfo`** – достъпни чрез инстанция.

- **`Create()`**, **`CreateSubdirectory()`** – създава директория или поддиректория.
- **`GetFiles(…)`** – връща всички файлове в директорията.
- **`GetDirectories(…)`** – връща всички поддиректории на директорията.
- **`MoveTo(…)`** – премества (преименува) директория.
- **`Delete()`** – изтрива директория.
- **`Exists()`** – проверява директория дали съществува.
- **`Parent`** – връща горната директория.
- **`FullName`** – пълно име на директорията.

Рекурсивно обхождане на директории – пример

За пример ще разгледаме програма, която обхожда дадена директория и извежда на конзолата нейното съдържание, като рекурсивно обхожда и поддиректориите в нея:

DirectoryTraversal.cs

```
using System;
using System.IO;

class DirectoryTraversal
{
   private static void Traverse(string aPath)
   {
      Console.WriteLine("[{0}]", aPath);
      string[] subdirs = Directory.GetDirectories(aPath);
      foreach (string subdir in subdirs)
      {
         Traverse(subdir);
      }

      string[] files = Directory.GetFiles(aPath);
      foreach (string f in files)
      {
         Console.WriteLine(f);
      }
   }

   static void Main()
   {
      string winDir = Environment.SystemDirectory;
      Traverse(winDir);
   }
}
```

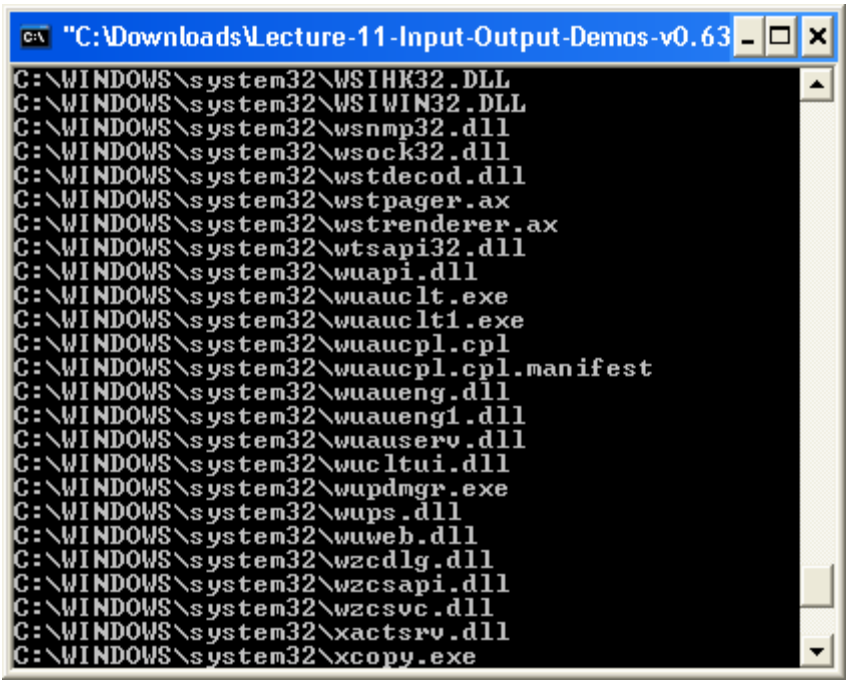
Как работи примерът?

Променливата **winDir** определя началната директория, от която започва обхождането. В случая, статичната член-променлива **SystemDirectory** на класа **Environment** определя директорията **C:\WINDOWS\system32** като начална.

Началната директория предаваме като параметър на рекурсивния метод **Traverse(…)**, който извършва обхождане в дълбочина. Той извежда подадената му директория на екрана, след което се самоизвиква за всяка една нейна поддиректория.

Поддиректориите на дадена директория се извличат с метода **Directory.GetDirectories(…)**, а файловете – с метода **Directory.GetFiles(…)**. И двата метода връщат като резултат масив от низове, съдържащи имена на директории или файлове, заедно с пълния път до тях.

Ето как би могъл да изглежда резултатът от изпълнението на горния пример:



Класът Path

Класът **System.IO.Path** предоставя допълнителна функционалност за работа с пътища. Той обработва **string** променливи, съдържащи информация за пътя до файл или директория. Функционалността, предоставена от класа **Path**, е независима от платформата. Ще изброим някои полезни свойства и методи:

- **DirectorySeparatorChar** – символът, който отделя директориите в пътя ("\" за Windows и "/" за UNIX и Linux файлови системи).
- **Combine(…)** – добавя относителен път към пълен.
- **GetExtension(…)** – извлича разширението на даден файл (ако има).
- **GetFileName(…)** – извлича име на файл от даден пълен път (ако има).
- **GetTempFileName(…)** – създава временен файл с уникално име и нулева дължина и връща името му.

Работа с временни файлове – пример

Следната кратка програма демонстрира работа с временен файл:

```
using System;
using System.IO;

class TempFilesDemo
```

```
{
   static void Main()
   {
      String tempFileName = Path.GetTempFileName();
      try
      {
         using (TextWriter writer =
            new StreamWriter(tempFileName))
         {
            writer.WriteLine("This is just a test");
         }
         File.Copy(tempFileName, "test.txt");
      }
      finally
      {
         File.Delete(tempFileName);
      }
   }
}
```

Как работи примерът?

Променливата **tempFileName** съдържа вече споменатия временен файл с уникално име. След като върху него отворим текстов писач и запишем текста "**This is just a test**", копираме временния файл в текстовия файл **test.txt**, който се създава в текущата директория (директорията **bin\Debug**) на приложението. След приключване на програмата, **test.txt** съдържа същия текст. Ако не изтрием временния файл във **finally** клаузата, можем да проверим, че върху хард диска остава новосъздаден файл с уникално име и разширение **.tmp**, който съдържа текста "**This is just a test**".

Специални директории

Специалните директории на текущия потребител са достъпни с метода **System.Environment.GetFolderPath(Environment.SpecialFolder)**. Ето и пример за достъп до някои от тях:

```
string myDocuments = Environment.GetFolderPath(
   Environment.SpecialFolder.Personal);
Console.WriteLine(myDocuments);
// C:\Documents and Settings\Administrator\My Documents

string myDesktop = Environment.GetFolderPath(
   Environment.SpecialFolder.DesktopDirectory);
Console.WriteLine(myDesktop);
// C:\Documents and Settings\Administrator\Desktop

string myFavourites = Environment.GetFolderPath(
```

```
  Environment.SpecialFolder.Favorites);
Console.WriteLine(myFavourites);
// C:\Documents and Settings\Administrator\Favorites

string myMusic = Environment.GetFolderPath(
  Environment.SpecialFolder.MyMusic);
Console.WriteLine(myMusic);
// C:\Documents and Settings\Administrator\My Documents\My Music
```

Наблюдение на файловата система

Класът **FileSystemWatcher** позволява наблюдение на файловата система за различни събития като създаване, промяна или преименуване на файл или директория. По-важните му събития и свойства са следните:

- **Path** – съдържа наблюдаваната директория.
- **Filter** – филтър за наблюдаваните файлове (напр. "***.***" или "***.exe**").
- **NotifyFilter** – филтър за типа на наблюдаваните събития, напр. **FileName**, **LastWrite**, **Size**.
- **Created**, **Changed**, **Renamed**, **Deleted** – събития, които се извикват при регистриране на промяна. Тези събития се извикват от друга нишка и кодът в тях трябва да е нишково-обезопасен, т.е. да не създава проблеми при конкурентен достъп до общи ресурси.

Наблюдение на файловата система – пример

Ще демонстрираме възможностите на класа **FileSystemWatcher** с един пример:

FileSystemWatcherDemo.cs

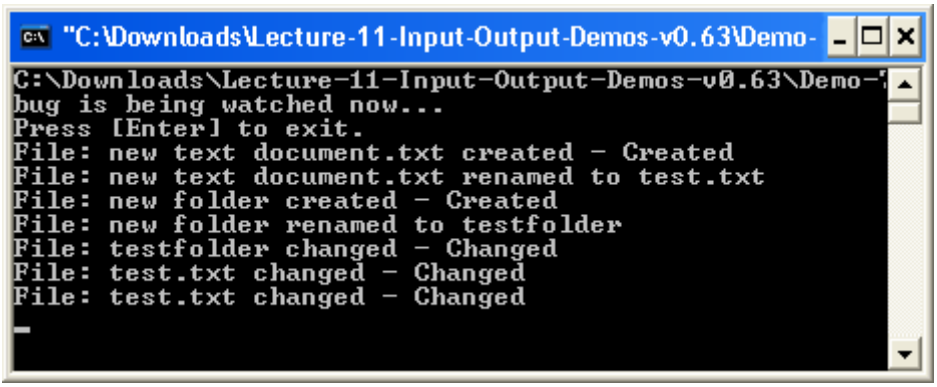
```
using System;
using System.IO;

class FileSystemWatcherDemo
{
  static void Main()
  {
    string currentDir = Environment.CurrentDirectory;

    FileSystemWatcher w =
      new FileSystemWatcher(currentDir);

    // Watch all files
    w.Filter = "*.*";
```

```
      // Watch the following information for the files
      w.NotifyFilter = NotifyFilters.FileName |
        NotifyFilters.DirectoryName |
        NotifyFilters.LastWrite;

      w.Created += new FileSystemEventHandler(OnCreated);
      w.Changed += new FileSystemEventHandler(OnChanged);
      w.Renamed += new RenamedEventHandler(OnRenamed);
      w.EnableRaisingEvents = true;

      Console.WriteLine(
        "{0} is being watched now...", currentDir);
      Console.WriteLine("Press [Enter] to exit.");

      Console.ReadLine();
   }

   // Methods called when a file is created, changed, or renamed
   private static void OnCreated(object aSource,
      FileSystemEventArgs aArgs)
   {
      Console.WriteLine("File: {0} created - {1}",
        aArgs.Name, aArgs.ChangeType);
   }

   private static void OnChanged(object aSource,
      FileSystemEventArgs aArgs)
   {
      Console.WriteLine("File: {0} changed - {1}",
        aArgs.Name, aArgs.ChangeType);
   }

   private static void OnRenamed(object aSource,
      RenamedEventArgs aArgs)
   {
      Console.WriteLine("File: {0} renamed to {1}",
        aArgs.OldName, aArgs.Name);
   }
}
```

Как работи примерът?

При конструирането на **`FileSystemWatcher`** обекта, му подаваме текущата директория – **`bin\Debug`** директорията на проекта. В тази директория ще наблюдаваме всички файлове, като ще следим изброените в **`w.NotifyFilter`** файлови операции. Събитията **`Created`**, **`Changed`** и **`Renamed`** свързваме с подходящи обработващи методи. Сега при създаване, модификация или преименуване на файл в наблюдаваната директория, се изпълнява съответният обработчик.

Ето примерен резултат от изпълнението на горната програма:



Това изпълнение на програмата отразява създаването на текстов файл в наблюдаваната директория и неговото преименуване. Следва създаване и преименуване на поддиректория. Накрая е показано какво се случва при промяна на съдържанието на файл в наблюдаваната директория.

Работа с IsolatedStorage

`IsolatedStorage` е технология, която се използва за приложения, които нямат достъп до локалния хард диск, но изискват локално съхраняване на файлове, напр. при приложения, стартирани с помощта на технологията .NET Zero Deployment, без да са зададени подходящи права за изпълнение. Технологията се използва и при работа с приложения, стартирани от Интернет, които по подразбиране работят с намалени права и не могат да осъществяват достъп до файловата система.

`IsolatedStorage` представлява виртуална файлова система. Тя е ограничена по обем и приложението, което я използва, няма достъп до останалите файлове на локалното устройство.

Класовете за достъп до изолирани файлови системи се намират в пространството от имена **`System.IO.IsolatedStorage`**.

Ако едно приложение просто съхранява данни в някакъв файл, този файл може лесно да бъде манипулиран от друго приложение или от друг потребител. Когато работи с **`IsolatedStorage`**, всяко приложение съхранява данни на място, уникално за него и за текущия потребител.

Нивата на изолация са две – първо, името на потребителя, стартирал приложението, и второ – името на стартираното асембли. В много случаи, името на асемблито съответства на URL адреса, откъдето то е било заредено и стартирано.

Данните, записвани на локалния диск на даден потребител, попадат в директорията му **`\Documents and Settings\<user>\Local Settings\Application Data\IsolatedStorage`**. При настройки по подразбиране за

всяка двойка (асембли, потребител) е зададено ограничение за обема на **IsolatedStorage** областта – 10 MB.

IsolatedStorage – пример

Следващият пример илюстрира четене и запис на текстов файл в областта **IsolatedStorage** за текущия потребител и асембли:

```
using System;
using System.IO;
using System.IO.IsolatedStorage;

class DirectoryTraversal
{
   static string ReadTextFileFromIsolatedStorage(
      string aFileName)
   {
      IsolatedStorageFile store = IsolatedStorageFile.GetStore(
         IsolatedStorageScope.User |
         IsolatedStorageScope.Assembly, null, null);
      using (store)
      {
         IsolatedStorageFileStream stream =
            new IsolatedStorageFileStream(aFileName,
            FileMode.Open, FileAccess.Read, store);
         using (stream)
         {
            StreamReader reader = new StreamReader(stream);
            using (reader)
            {
               string result = reader.ReadToEnd();
               return result;
            }
         }
      }
   }

   static void WriteTextFileToIsolatedStorage(
      string aFileName, string aText)
   {
      IsolatedStorageFile store = IsolatedStorageFile.GetStore(
         IsolatedStorageScope.User |
         IsolatedStorageScope.Assembly, null, null);
      using (store)
      {
         IsolatedStorageFileStream stream =
            new IsolatedStorageFileStream(aFileName,
            FileMode.Create, FileAccess.Write, store);
         using (stream)
         {
```

```
            StreamWriter writer = new StreamWriter(stream);
            using (writer)
            {
               writer.Write(aText);
            }
         }
      }
   }

   static void Main()
   {
      try
      {
         string text=ReadTextFileFromIsolatedStorage("notes.txt");
         Console.WriteLine("Text read from isolated storage: {0}",
            text);
      }
      catch (IOException ioex)
      {
         Console.WriteLine(
            "Error reading from isolated storage: {0}", ioex);
      }

      try
      {
         string text = "Just a test!";
         WriteTextFileToIsolatedStorage("notes.txt", text);
         Console.WriteLine(
            "Text written to isolated storage: {0}", text);
      }
      catch (IOException ioex)
      {
         Console.WriteLine(
            "Error writing to isolated storage: {0}", ioex);
      }
   }
}
```

Повече внимание на технологията **`IsolatedStorage`** ще обърнем в темата "Сигурност в .NET Framework".

Упражнения

1. Какво представляват потоците в .NET Framework? Кои са основните операции с тях? Кои са основните класове за работа с потоци?
2. Напишете програма, която кодира двоични файлове по зададена ключова фраза (символен низ). Използвайте следния алгоритъм за кодиране: кодирайте първия байт от файла с операцията "изключващо или" с първия байт от ключовата фраза. Вторият байт от файла коди-

райте с втория байт от ключовата фраза и т.н. При достигане на последния байт преминавайте към първия. Използвайте файлови потоци. Напишете и програма за декодиране на така кодираните файлове.

3. Напишете програма, която съхранява данни за студенти във файл със записи. За всеки студент трябва да се съхранява неговото име, факултетен номер, курс и среден успех. Реализирайте методи за добавяне на студент, за търсене на студент по име и по факултетен номер, за сортиране на студентите по успех и за отпечатване на всички студенти от файла. Използвайте бинарни четци и писачи.
4. Напишете програма, която обръща на обратно бинарен файл по следния начин: първият му байт става последен, вторият става предпоследен и т.н. Използвайте файлови потоци без да използвате временни файлове. Помислихте ли за ефективността? Приемлива ли е скоростта на работа при файлове с размер 700 MB?
5. Напишете програма, която разделя даден двоичен файл на еднакви части (файлове с фиксиран размер, примерно 1.44 MB). Файловете трябва да се номерират автоматично. Използвайте файлови потоци.
6. Напишете програма, която съединява частите, генерирани от предходната програма и възстановява оригиналния файл, от който са получени. Използвайте файлови потоци.
7. Напишете програма, която по даден списък от нецензурни думички заменя в текстов файл всяко срещане на думичка от списъка със звездички (със същия брой букви). Списъкът от нецензурни думички трябва да се прочете от текстов файл **`words.txt`**. Използвайте временен изходен файл и след като получите резултата в него, изтрийте входния файл и преименувайте временния файл с името на изтрития вече входен файл. Използвайте текстови четци и писачи.
8. Напишете програма, която търси даден символен низ във всички текстови файлове (**`*.txt`**) от дадена директория и нейните поддиректории. При всяко съвпадение трябва да се отпечатва пълното име на файла, където е намерено съвпадението, номерът на реда в този файл и съдържанието на този ред. Използвайте текстов четец за прочитане на файловете ред по ред.
9. Напишете програма, която намира всички текстови файлове (**`*.txt`**) на твърдия диск като извършва обхождане в ширина чрез опашка по следния начин: 1. добавя в опашката началната директория. 2. докато опашката не остане празна изважда от нея директорията, която е влязла най-рано, намира и отпечатва всички файлове от нея и добавя в опашката всичките поддиректории на текущата.
10. Напишете програма, която съхранява данни за студенти във файл със записи. За всеки студент трябва да се съхранява неговото име, факултетен номер, курс и среден успех. Реализирайте методи за добавяне на студент, за търсене на студент по име и по факултетен номер, за

сортиране на студентите по успех и за отпечатване на всички студенти от файла. Използвайте бинарни четци и писачи.

11. Напишете програма, която следи даден текстов файл от дадена директория и при промяна на съдържанието му го отпечатва на конзолата.

12. Напишете програма, която при първо стартиране пита потребителя за името му и го записва в текстов файл в IsolatedStorage областта. При следващо стартиране програмата трябва да го поздравява с името, прочетено от IsolatedStorage областта и да му дава възможност да го промени.

Използвана литература

1. Светлин Наков, Вход и изход в .NET Framework – http://www.nakov.com/dotnet/lectures/Lecture-11-Input-Output-v1.0.ppt
2. MSDN Library – http://msdn.microsoft.com
3. Inside C#, 2nd Edition, Tom Archer, Andrew Whitechapel
4. Стоян Йорданов, Потоци и файлове – http://www.nakov.com/dotnet/2003/lectures/Streams-and-Files.doc
5. Георги Иванов, Потоци и файлове – http://www.nakov.com/2003/dotnet/lectures/Streams-and-Files.doc
6. MSDN Training, Programming with the Microsoft® .NET Framework (MOC 2349B), Module 10: Data Streams and Files
7. Светлин Наков, Интернет програмиране с Java, Фабер, 2004, ISBN 954-775-305-3, тема 1.2 (Вход/изход с Java)

Глава 12. Работа с XML

Необходими знания

- Базови познания за .NET Framework
- Базови познания за езика C#
- Базови познания за езика XML и свързаните с него технологии

Съдържание

- Какво е XML?
- XML и HTML
- Кога се използва XML?
- Пространства от имена
- Схеми и валидация – DTD, XSD и XDR схеми
- Редакторът за схеми на VS.NET
- XML парсери
- XML поддръжка в .NET Framework
- Работа с DOM парсера – класовете **`XmlNode`** и **`XmlDocument`**
- SAX парсери и класът **`XmlReader`**
- Кога да използваме DOM и кога SAX?
- Създаване на XML документи с **`XmlWriter`**
- Валидация на XML по схема
- Работа с XPath – класовете **`XPathNavigator`** и **`XPathDocument`**
- XSL трансформации в .NET Framework

В тази тема...

В настоящата тема ще разгледаме работата с XML в .NET Framework. Ще обясним накратко какво представлява езикът XML. Ще обърнем внимание на приликите и разликите между него и HTML. Ще разгледаме какви са приложенията на XML. Ще се запознаем с пространствата от имена в XML и различните схеми за валидация на XML документи (DTD, XSD, XDR), като ще представим и средствата на Visual Studio .NET за работа с XSD схеми. Ще разгледаме особеностите на класическите XML парсери (DOM и SAX) и как те са имплементирани в .NET Framework. Ще опишем подробно класовете за работа с DOM парсера (**`XmlNode`** и **`XmlDocument`**) и ролята на класа **`XmlReader`** за SAX парсерите в .NET Framework. Ще опишем ситуациите, при които е подходяща употребата на DOM или SAX модела. Ще се

запознаем с начина на работа на класа **`XmlWriter`** за създаване на XML документи. Ще разгледаме начините за валидация на XML документи спрямо дадена схема с помощта на валидиращи парсери. Ще представим поддръжката в .NET Framework и на някои други XML-базирани технологии като XPath и XSLT.

Какво е XML?

Преди да преминем към класовете, които .NET Framework предоставя за работа с XML, нека първо разгледаме същността на тази технология.

XML (Extensible Markup Language)

XML първоначално е замислен като език за дефиниране на нови документни формати за World Wide Web. XML произлиза от SGML (Standard Generalized Markup Language) и на практика е негово подмножество със значително опростен синтаксис, което прави внедряването му много полесно. С течение на времето XML се налага като markup език за структурирана информация.

Какво представлява един markup език?

Произходът на термина **markup** е свързан с областта на печатните издания, но при електронните документи markup описва специфичното обозначаване на части от документите с тагове. Таговете имат две основни предназначения – те описват изгледа и форматирането на текста или определят структурата и значението му (метаинформация).

Днес се използват два основни класа markup езици – специализирани и с общо предназначение (generalized) markup езици. Първата група езици служат за генериране на код, който е специфичен за определено приложение или устройство и адресира точно определена необходимост. Общите markup езици описват структурата и значението на документа, без да налагат условия по какъв начин ще се използва това описание. Пример за специализиран markup език е HTML, докато SGML и неговото функционално подмножество XML са типични markup езици с общо предназначение.

XML markup – пример

Следният пример демонстрира концепцията на markup езиците с общо предназначение. При тях структурата на документа е ясно определена, таговете описват съдържанието си, а форматирането и представянето на документа не е засегнато – всяко приложение може да визуализира и обработва XML данните по подходящ за него начин.

```
<?xml version="1.0" encoding="windows-1251"?>
<messages>
  <message>XML markup описва структура и съдържание</message>
  <message>XML markup не описва форматиране</message>
</messages>
```

Универсална нотация за описание на структурирани данни

XML представлява набор от правила за съставяне на текстово-базирани формати, които улесняват структурирането на данни. Една от характеристиките, които налагат XML като универсален формат, е възможността да представя както структурирана, така и полуструктурирана информация. XML има отлична поддръжка на интернационализация, благодарение на съвместимостта си с Unicode стандарта. Друг универсален аспект на XML е способността му да отделя данните от начина им на представяне.

XML съдържа метаинформация за данните

XML спецификацията определя стандартен начин за добавяне на markup (метаинформация) към документите. Метаинформацията представлява информация за самата информация и нейната структура – така се осъществява връзката между данните, представени в XML документа, и тяхната семантика.

XML е метаезик

XML е метаезик за описание на markup езици. Той няма собствена семантика и не определя фиксирано множество от тагове. Разработчикът на едно XML приложение има свободата да дефинира подходящо за конкретната ситуация множество от XML елементи и евентуални структурни връзки между тях.

XML е световно утвърден стандарт

XML е световно утвърден стандарт, поддържан от W3C (World Wide Web Consortium, http://www.w3.org). XML не е самостоятелна технология, а по–скоро е основа на цяла фамилия от XML-базирани технологии като XPath, XPointer, XSLT и др., които също се поддържат и развиват от W3C.

XML е независим

Езикът XML е независим от платформата, езиците за програмиране и операционната система. Тази необвързаност го прави много полезен при нужда от взаимодействие между хетерогенни програмни платформи и/или операционни системи.

XML – пример

Следният кратък пример демонстрира един възможен начин за описание на книгите в една библиотека със средствата на XML. Информацията е лесно четима и разбираема, самодокументираща се и технологично-независима.

```
<?xml version="1.0"?>
<library name=".NET Developer's Library">
  <book>
    <title>Programming Microsoft .NET</title>
    <author>Jeff Prosise</author>
    <isbn>0-7356-1376-1</isbn>
  </book>
  <book>
    <title>Microsoft .NET for Programmers</title>
    <author>Fergal Grimes</author>
    <isbn>1-930110-19-7</isbn>
  </book>
</library>
```

XML и HTML

Външно езикът XML прилича на езика HTML, но между двата езика има и сериозни различия.

Прилики между езиците XML и HTML

XML и HTML си приличат по това, че са текстово-базирани и използват тагове и атрибути.

Текстово-базирани

XML и HTML са текстово-базирани езици и това осигурява прозрачност на информационния формат. При нужда такива документи могат да се отварят и редактират с помощта на обикновен текстов редактор.

Използват тагове и атрибути

Двата езика използват елементи, всеки от които се състои от отварящ и затварящ таг (например **<book>** и **</book>**) и информация между тях (представяща съдържанието на елемента). Всеки елемент може да дефинира свои атрибути, които съдържат метаданни за съдържанието му.

Разлики между езиците XML и HTML

XML и HTML си приличат само външно. Те имат съвсем различно предназначение и това води до някои сериозни различия.

HTML е език, а XML – метаезик

Въпреки че и двата езика произлизат от SGML, на практика HTML е негово специализирано приложение, докато XML е функционално подмножество на SGML. HTML елементите и атрибутите са предефинирани и с ясно определен смисъл. XML от своя страна запазва гъвкавостта и разширяемостта на SGML - той не дефинира собствена семантика и набор от тагове, а предоставя синтаксис за описание на други езици.

HTML описва форматиране, а XML – структурирана информация

HTML е проектиран с единствената цел да осигури начин за форматиране на документи в World Wide Web. Той не е разширяем и не поддържа произволни структури от данни. За разлика от него XML предоставя средства за дефиниране на произволни тагове и структурни връзки между тях. HTML описва как да се представи информацията, докато XML описва самата информация, като я структурира по стандартен начин, разбираем за различни приложения.

HTML, XML и добре дефинираните документи

Въпреки че XML и HTML документите си приличат на външен вид (с тази разлика, че таговете на единия език са предефинирани, а на другия – не), XML синтаксисът е много по-строг и не допуска отклонения за разлика от HTML. В един HTML документ е допустима употребата на некоректно зададени тагове и те се игнорират впоследствие от браузъра, ако той не намери начин как да ги обработи. В XML спецификацията изрично се забранява на приложенията, обработващи XML документи, да гадаят смисъла на синтактично некоректен файл. Ако XML документът не е добре дефиниран, обработката му трябва да се прекрати и да се докладва за грешка.

Добре дефинирани документи

Езикът XML дефинира понятието "добре дефинирани документи" (well-formed documents). Да разгледаме какво точно означава това.

XML изисква добре дефинирани документи

Някои основни правила, който определят един XML документ като добре дефиниран, са следните:

- документът да има само един основен документен елемент
- таговете винаги да се затварят и то в правилен ред (да не се застъпват)
- атрибутите винаги да се затварят по правилен начин
- имената на таговете и атрибутите да отговарят на някои ограничения

Правете разлика между коренен ("/") и документен елемент (<library> в горния пример) в един XML документ. Тези понятия не са синоними!

Пример за лошо дефиниран XML документ

В следващия пример са нарушени почти всички правила, изброени по-горе – отварящи и затварящи тагове не си съответстват, тагове не се

затварят, не са спазени ограниченията за името на атрибута **bug!** и атрибутът **value** не е коректно затворен:

```
<xml>
    <button bug! value="OK name="b1">
        <animation source="demo1.avi"> 1 < 2 < 3
    </click-button>
< / xml >
```

Кога се използва XML?

Езикът XML има изключително широка употреба в съвременните софтуерни технологии, защото предоставя универсален формат за съхранение и обмен на информация, а от това имат нужда болшинството от съвременните софтуерни системи.

Обмяна на информация

Обмяната на информация между системи, които боравят с несъвместими формати, е сериозно предизвикателство в съвременното информационно общество. Много системи работят с нестандартизирани, собствени формати и при нужда от взаимодействие разработчиците трябва да полагат много усилия, за да осигурят съвместимост на обменяните данни при комуникацията.

XML е едно възможно решение на този проблем, тъй като позволява дефинирането на специфичен за приложението, прозрачен формат за обмяна на информация. XML документите, които спазват публикуваните от W3C спецификации, са разбираеми за всички приложения и така XML може да се използва като общ междинен формат при обмяната на информация.

Съхранение на структурирани данни

Почти всяко приложение има нужда от съхранение на данни. В много случаи XML е подходящ за тази задача, тъй като разделя структурираната информация от нейното визуално представяне. XML е подходящ формат за съхранение най-вече на малки информационни файлове или на данни, които не се очаква да поддържат произволно търсене (достъп). XML markup описва структурата на данните наред с тяхното съдържание. Това позволява да се дефинират схеми за валидация на XML документи, чрез които да се установява валидността на XML структурата.

Недостатъци на XML

Наред с предимствата, които предоставя, XML понякога става причина за значително увеличение на размера на данните и времето, необходимо за обработката им.

Обемисти данни

XML е текстово-базиран формат, който използва тагове като ограничители (и описатели) на съдържаната в документа информация. Самата му природа (текстов формат с чести и повтарящи се етикети) е предпоставка за увеличен размер на файловете (съответно и увеличен мрежов трафик). Големината на един XML файл винаги е по-голяма от тази на файл със същата информация, записана в сравним двоичен формат.

Този недостатък обикновено може да бъде компенсиран на други нива. Дисковото пространство днес е далеч по-евтино, а алгоритмите за компресия позволяват бърза и качествена компресия при нужда (особено при текстови данни). Комуникационните протоколи като HTTP/1.1 могат да компресират информацията "в движение", спестявайки мрежов трафик толкова ефективно, колкото и при употребата на двоичен формат.

Повишена необходимост от физическа памет

Един XML документ може да бъде голям по размер по два критерия – в статичния си файлов формат (нужда от повече дисково пространство за съхранение) или в заредената в динамичната памет форма (нужда от повече изчислителни ресурси и RAM памет). Като пряко следствие от това, че XML данните са значителни по обем, идва повишената необходимост от физическа памет за съхраняването им.

Съвместимостта на XML с Unicode кодовата таблица също указва влияние. Например **`short int`** стойността 12355 има текстово представяне между 5 и 20 байта и само 2 байта, ако бъде съхранявана в двоична форма.

Намалена производителност

Заслужено или не, XML си е създал репутацията на технология, "лакома" за ресурси. XML се записва като текст и XML данните са в абстрактен логически формат, описващ тяхната структура. За прочитането им в едно приложение често са нужни две стъпки - парсване на XML информацията от нейния текстов вид и преобразуването на така получените данни, за да станат използваеми от страна на приложението. Парсването и трансформацията изискват време, а същото важи и за генерирането на изходящ XML поток. Въпреки това, най-сериозната опасност за производителността идва от способността на XML да включва и зарежда външни ресурси (DTD файлове, XSD схеми).

Пространства от имена

Пространствата от имена представляват логически свързани множества, които изискват всички принадлежащи им имена да са уникални. Те служат за различаване на елементите и атрибутите от различни XML приложения, които притежават еднакви имена. Пространствата от имена групират всички свързани елементи и атрибути от едно XML приложение и улесняват разпознаването им от страна на софтуера.

Дефиниране на пространства от имена

Имената на елементите и атрибутите се състоят от две части – име на пространството, на което принадлежат, и локално име. Това съставно име е известно като квалифицирано име (qualified name, Qname). Идентификаторите на пространствата от имена в XML трябва да се придържат към специфичен URI (Uniform Resource Identifier) синтаксис. URI спецификацията дефинира две основни URI форми – URL (Uniform Resource Locators) например **http://www.nakov.com/town** и URN (Uniform Resource Names) например **urn:nakov-com:country**.

URI идентификаторите на пространствата от имена не подлежат на анализ от страна на процесора – те са единствено средство за идентификация и няма изискване да сочат към реално достъпни ресурси в мрежата.

URI идентификаторите обикновено са доста дълги и вместо тях в XML документите се използва префикс за асоцииране на локалните елементи и атрибути с определено пространство от имена. Префиксът е просто съкратен псевдоним за един URI идентификатор, който се свързва с него при дефинирането на пространство от имена:

```
xmlns:<префикс>="<идентификатор на пространство от имена>"
```

Използване на тагове с еднакви имена – пример

Следващият пример демонстрира как пространствата от имена разрешават двусмислието при използване на тагове с еднакви имена в един XML документ:

```
<?xml version="1.0" encoding="UTF-8"?>
<country:towns xmlns:country="urn:nakov-com:country"
     xmlns:town="http://www.nakov.com/town">
  <town:town>
    <town:name>Sofia</town:name>
    <town:population>1 200 000</town:population>
    <country:name>Bulgaria</country:name>
  </town:town>
  <town:town>
    <town:name>Plovdiv</town:name>
    <town:population>700 000</town:population>
    <country:name>Bulgaria</country:name>
  </town:town>
</country:towns>
```

Дефинирани са две пространства от имена: **urn:nakov-com:country** с префикс **country** и **http://www.nakov.com/town** с префикс **town**. Всяко от тези пространства съдържа елемент **<name>**, но проблем не съществува,

защото елементите са определени от префикса на собственото си пространство от имена – **<country:name>** описва името на държавата, докато **<town:name>** съдържа името на града.

Пространства по подразбиране

Използването на префиксно-ориентиран синтаксис е сравнително интуитивен процес за повечето софтуерни разработчици. Съществува обаче и друг начин за асоцииране на XML елементите с пространствата от имена – дефинирането на пространства по подразбиране. Използва се следният синтаксис:

```
xmlns="<идентификатор на пространство от имена>"
```

Пространствата по подразбиране не използват префикси. При дефиниране на такова пространство в един XML елемент всички неасоциирани с префикс (или друго пространство от имена) елементи в неговия обсег на видимост автоматично се свързват с пространството по подразбиране. Възможно е декларацията на пространство по подразбиране да бъде отменена – за целта на идентификатора му се присвоява празен низ:

```
<language xmlns="">C#</language>
```

Пространства по подразбиране – пример

Примерът демонстрира дефинирането на пространство по подразбиране **http://www.hranitelni-stoki.com/orders**. Елементът **<item>** не е изрично асоцииран с пространство от имена, затова той автоматично се свързва с пространството по подразбиране. Пълното име на елемента **<item>** е **http://www.hranitelni-stoki.com/orders:item**.

```
<?xml version="1.0" encoding="windows-1251"?>
<order xmlns="http://www.hranitelni-stoki.com/orders">
  <item>
    <name>бира "Загорка"</name>
    <amount>8</amount>
    <measure>бутилка</measure>
    <price>3.76</price>
  </item>
  <item>
    <name>кебапчета</name>
    <amount>12</amount>
    <measure>брой</measure>
    <price>4.20</price>
  </item>
</order>
```

Пространства от имена и пространства по подразбиране – пример

Следващият пример демонстрира един възможен начин на съвместна употреба на пространства по подразбиране и други пространства от имена:

```
<?xml version="1.0" encoding="utf-8" ?>
<faculty:student xmlns:faculty="urn:fmi"
     xmlns="urn:foo" id="235329">
  <name>Ivan Ivanov</name>
  <language xmlns="">C#</language>
  <rating>6.00</rating>
</faculty:student>
```

Дефинирани са пространство от имена **urn:fmi** с префикс **faculty** и пространство по подразбиране **urn:foo**. Елементът **student** принадлежи на пространството от имена с идентификатор **urn:fmi**, докато елементите **name** и **rating** са от пространството по подразбиране **urn:foo**. Елементът **language** от друга страна не принадлежи на нито едно пространство от имена, тъй като за него пространството по подразбиране е отменено.

В крайна сметка пълните имена на елементите от примерния документ са съответно **urn:fmi:student**, **urn:foo:name**, **urn:foo:rating** и **language**.

Автоматичното асоцииране с пространството по подразбиране на елементи, несвързани с друго пространство, не се отнася за атрибутите. Поради тази причина атрибутът `id` в горния пример не принадлежи на нито едно пространство от имена.

За разлика от пространството по подразбиране, префиксните пространства не могат да се отменят.

Схеми и валидация

Обичайно под думата схема се разбира общо представяне на даден клас предмети. В смисъла на XML, **схема** е формално описание на формата на XML документи.

Документ, който издържа успешно теста, описан от съответната XML схема, се определя като валиден (съобразяващ се със схемата). Процесът на тестване на документ спрямо зададена схема се нарича **валидация**.

Схемата гарантира, че документът изпълнява определени изисквания. Тя открива грешки в документа, които в последствие могат да доведат до неправилната му обработка. Схемите лесно се публикуват в Интернет и могат да служат като общодостъпен начин за описание на синтаксиса на дадено XML приложение.

XML схеми – защо са необходими?

Пространствата от имена дефинират синтаксис за групиране на свързани елементи от едно XML приложение и начин за обръщение към тях, но не разглеждат въпроса кои са тези елементи. Съдържанието на XML документите се контролира чрез дефиниране на схеми. Схемите контролират структурата на XML документите и дефинират необходимия синтаксис за целта. Схемите описват:

- допустими тагове, които могат да присъстват в един XML документ
- допустими атрибути за тези тагове
- допустими стойности за елементите и атрибутите в документа
- ред на поставянето на таговете в XML документа
- дефинират стойности по подразбиране

XML схеми – видове

Съществуват различни видове XML схеми, като всяка има своите силни и слаби страни. Ще разгледаме особеностите на няколко от най-популярните стандарти за XML схеми – DTD, XSD и XDR.

Езикът DTD

DTD (Document Type Definition) е формален език за описание структурата на XML документи. Този език е оригиналният XML документен модел, той присъства и в XML спецификацията. DTD всъщност датира отпреди времето на XML – DTD произтича от SGML стандартите, като основният синтаксис е запазен почти изцяло.

DTD съдържа правила за таговете и атрибутите в документа

DTD контролира структурата на един XML документ, като дефинира множество от разрешени за използване елементи. Други елементи извън описаните не могат да присъстват в документа.

Езикът дефинира модел на съдържание (content model) за всеки елемент. Този модел определя допустимите елементи или данни, които могат да се съдържат в един елемент, наредбата и броя им, а също и дали присъствието на определен елемент е задължително или избирателно.

DTD декларира множество от позволени атрибути за всеки елемент. Декларация на атрибут определя името, типа данни, стойностите по подразбиране (ако има такива) и указва дали атрибутът задължително трябва да присъства в документа или не.

Текстово-базиран език, но не е XML

DTD е текстово-базиран език, който е запазил в основна степен синтаксиса на SGML. DTD обаче не е XML-базиран стандарт – разработен е преди

появата на XML и днес малко по-малко отстъпва позициите си пред XML-базирани стандарти за схеми като XSD.

Дефиниране на DTD – пример

Следващият пример демонстрира една възможна DTD декларация, която контролира съдържанието на малка домашна библиотека:

library.dtd

```
<!-- contents of library.dtd -->
<!ELEMENT library (book+)>
<!ATTLIST library name CDATA #REQUIRED>
<!ELEMENT book (title, author, isbn)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT isbn (#PCDATA)>
```

Дефиниран е елемент с име **library**, който съдържа една или повече (но поне една) инстанция на елемента **book**. За елемента **library** е дефиниран списък от атрибути – **library** задължително трябва да притежава атрибут с име **name** от тип **CDATA** (**character data**).

Забележете, че DTD не определя **library** за документен елемент – DTD декларацията не може да разграничи никой елементите като кандидат за документен елемент в XML документа. Елементът **book** трябва да съдържа в себе си точно по една инстанция на елементите **title**, **author** и **isbn**. Те от своя страна са дефинирани като елементи от тип **PCDATA** (**parsed character data**).

Данните, дефинирани като CDATA (character data), не се обработват от парсера. Текстът в рамките на CDATA не се третира като markup, а се разглежда като чист текст. PCDATA (parsed character data) елементите подлежат на парсване и съдържанието им се третира като нормален markup.

Използване на DTD – пример

След като сме дефинирали горното DTD описание, лесно можем да го асоцииираме с даден XML документ и след това да го използваме при валидация. За целта в този документ вмъкваме **DOCTYPE** декларация, която указва името на документния елемент (в този случай **library**) и относителния път до самата DTD декларация:

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE library PUBLIC "library.dtd">
<library name=".NET Developer's Library">
  <book>
```

```
    <title>Programming Microsoft .NET</title>
    <author>Jeff Prosise</author>
    <isbn>0-7356-1376-1</isbn>
  </book>
  <book>
    <title>Microsoft .NET for Programmers</title>
    <author>Fergal Grimes</author>
    <isbn>1-930110-19-7</isbn>
  </book>
</library>
```

Ключовата дума **PUBLIC** обозначава факта, че работим с публична външна DTD декларация (другата възможност е да се използва ключовата дума **SYSTEM** за частни DTD декларации).

XSD схеми

Въпреки че DTD присъства в оригиналната XML спецификация, с течение на времето става ясно, че е необходим по-мощен инструмент за описание на XML документите и това води до появата на езика XSD (XML Schema Definition Language).

Мощен XML-базиран език за описание на XML структурата

За разлика от DTD, XSD е XML-базиран език за описване структурата на XML документи. Той, подобно на DTD, съдържа съвкупност от правила за таговете в документа и техните атрибути. Езикът XSD осигурява система от типове за XML обработка, която е много по-силно типизирана, отколкото DTD.

Вградени типове данни

XML Schema предоставя набор от вградени типове данни, които разработчиците могат да използват, за да ограничават съдържанието на текста. Тези типове данни са описани в пространството от имена **http://www.w3.org/2001/XMLSchema**. Всеки от тях има дефинирана област от допустими стойности.

Потребителски-дефинирани типове данни

Към набора от предефинирани типове данни, XSD позволява употребата и на потребителски типове. XSD поддържа дефинирането на два основни потребителски класа – прости типове (чрез таг **xs:simpleType**, където **xs** е префикс за системното пространство от имена **http://www.w3.org/2001/XMLSchema**) и комплексни типове (чрез таг **xs:complexType**).

Простите типове не задават структура, а само стойностно поле, и могат да бъдат задавани само на текстови елементи (без наследници) и атрибути.

Елементите, притежаващи допълнителна структура – например с дефинирани атрибути или наследници - трябва да бъдат описани с комплексен тип.

Дефиниране на XSD схеми – пример

Настоящият пример демонстрира един възможен начин за дефиниране на XSD схема за валидация на съдържанието на малка домашна библиотека:

library.xsd

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="library">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="book" maxOccurs="unbounded" />
      </xs:sequence>
      <xs:attribute name="name" type="xs:string"
            use="optional" />
    </xs:complexType>
  </xs:element>
  <xs:element name="book">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="title" />
        <xs:element ref="author" />
        <xs:element ref="isbn" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="title" type="xs:string" />
  <xs:element name="author" type="xs:string" />
  <xs:element name="isbn" type="xs:string" />
</xs:schema>
```

Схемата дефинира пет глобални елемента – **library**, **book**, **title**, **author** и **isbn** – като три от тях са дефинирани от тип **string** (**xs:string**, където **xs** отново е префикс за пространството от имена на **XMLSchema**), а **library** и **book** са дефинирани като комплексни типове. Всеки един от глобалните елементи може да бъде използван като документен елемент в XML файл. Структурата на **library** определя, че този елемент съдържа неограничен брой елементи **book** и има незадължителен атрибут **name** от тип **string**. Елементът **book** е съставен от **title**, **author** и **isbn** (точно в тази последователност) и не дефинира атрибути.

Използване на XSD схеми – пример

Гореописаната XSD схема лесно може да се асоциира с даден XML документи и да се използва за неговата валидация. Това става най-лесно

с помощта на дефинирания в пространството от имена **http://www.w3.org/2001/XMLSchema-instance** атрибут **noNamespaceSchemaLocation**, който указва относителния път до XSD документа, съдържащ съответната валидираща схема:

```
<?xml version="1.0" encoding="utf-8" ?>
<library name=".NET Developer's Library"
     xsi:noNamespaceSchemaLocation="library.xsd"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <book>
     <title>Programming Microsoft .NET</title>
     <author>Jeff Prosise</author>
     <isbn>0-7356-1376-1</isbn>
   </book>
   <book>
     <title>Microsoft .NET for Programmers</title>
     <author>Fergal Grimes</author>
     <isbn>1-930110-19-7</isbn>
   </book>
</library>
```

Освен **xsi:noNamespaceSchemaLocation** за адресиране на XSD схеми се използва и атрибутът **xsi:schemaLocation** – за схеми, свързани с определени пространства от имена.

XSD измества DTD

Един от най-големите недостатъци на DTD е липсата на поддръжка на пространства от имена, тъй като те са въведени по-късно. DTD изисква всеки елемент в един XML документ да има съответна декларация в DTD файла, което противоречи на идеята за XML пространствата от имена.

Друг съществен проблем пред DTD е слабо типизираната система от типове, която се прилага само за дефинираните атрибути. DTD е насочен главно към описание на структурата на един документ и обръща много по-малко внимание на съдържанието извън това дали елементите могат да съдържат character data или не. Единствено атрибутите могат да бъдат декларирани от различни типове (**ID**, **IDREF**, **enumerated**), но ограничения върху типа данни в един елемент не могат да бъдат налагани.

За справянето с такива проблеми е разработена нова система за описание на структурата и валидация на XML документите – XSD схеми. Поголямата изразителна мощ на XSD в сравнение с DTD декларациите води до постепенно налагане на XSD схемите като основно средство за валидация на документи.

XDR схеми

XDR (XML-Data Reduced) е още един XML-базиран език за описание структурата на XML документи. Той е въведен от Microsoft преди появата

на стандартизираните от W3C XSD схеми. XDR е представен през 1999 като работна схема за валидация в продукта Microsoft BizTalk Server. XDR схемите са компактен вариант на XML-Data схемите. Те са по-мощни от DTD декларациите, но същевременно са по-слабо изразителни от XSD схемите, които се появяват през 2001.

В последно време XDR схемите губят своята популярност дори и при Microsoft-базираните продукти и технологии, където традиционно са намирали своето приложение (BizTalk, SQL Server 2000). XDR поддържа типове от данни и пространства от имена. Интересно е, че тези схеми могат да описват съответствия между структурата на XML документи и релационни бази данни.

Декларация на XDR схеми – пример

Следният пример демонстрира един възможен начин за дефиниране на XDR схема за валидация на малка домашна библиотека:

library.xdr

```
<?xml version="1.0" encoding="UTF-8"?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
     xmlns:dt="urn:schemas-microsoft-com:datatypes">
   <ElementType name="author" model="closed"
     content="textOnly" dt:type="string"/>
   <ElementType name="title" model="closed"
     content="textOnly" dt:type="string"/>
   <ElementType name="isbn" model="closed"
     content="textOnly" dt:type="string"/>
   <ElementType name="book" model="closed"
     content="eltOnly" order="seq">
     <element type="title" minOccurs="1" maxOccurs="1"/>
     <element type="author" minOccurs="1" maxOccurs="1"/>
     <element type="isbn" minOccurs="1" maxOccurs="1"/>
   </ElementType>
   <ElementType name="library" model="closed"
     content="eltOnly" order="seq">
     <AttributeType name="name" dt:type="string"
       required="yes"/>
       <attribute type="name"/>
     <AttributeType name="xmlns" dt:type="string"/>
       <attribute type="xmlns"/>
     <element type="book" minOccurs="1" maxOccurs="*"/>
   </ElementType>
</Schema>
```

Схемата определя, че съдържанието на елементите **author**, **title** и **isbn** може да бъде единствено текст, но не и други елементи (**content="textOnly"**). Стойността на атрибута **model** (**closed**) показва, че тези елементи не могат да съдържат елементи и атрибути, освен изрично

споменатите в модела на съдържанието (content model) на съответния елемент (такива в този случай също няма).

Елементът **book** от своя страна не може да съдържа свободен текст, а само елементите, описани в неговия модел на съдържанието (**content="eltOnly"**), като те трябва да спазват точната последователност (**order="seq"**) – точно по един елемент в реда **title**, **author** и **isbn**.

Последният дефиниран елемент **library** може да съдържа неограничен брой елементи **book** (но най-малко един) и има задължителен атрибут **name** и незадължителен атрибут **xmlns**.

Ако декларацията за незадължителен атрибут `xmlns` не присъстваше, при определения `model="closed"` нямаше да е възможно да включваме други пространства от имена към XML документа, валидиран от тази XDR схема.

Друга възможност е да се използва `model="open"` и тогава елементи и атрибути, независимо че не са декларирани изрично в модела на съдържанието на даден елемент, могат да бъдат добавяни към него.

Използване на XDR схеми – пример

Така декларираната по-горе XDR схема можем без много усилия да приложим за валидация на XML документ. Необходимо е в документния му елемент да се съдържа специално форматиран атрибут за включване на пространство от имена:

```
<?xml version="1.0"?>
<library name=".NET Developer's Library"
     xmlns="x-schema:http://url-of-schema/library.xdr">
   <book>
     <title>Programming Microsoft .NET</title>
     <author>Jeff Prosise</author>
     <isbn>0-7356-1376-1</isbn>
   </book>
   <book>
     <title>Microsoft .NET for Programmers</title>
     <author>Fergal Grimes</author>
     <isbn>1-930110-19-7</isbn>
   </book>
</library>
```

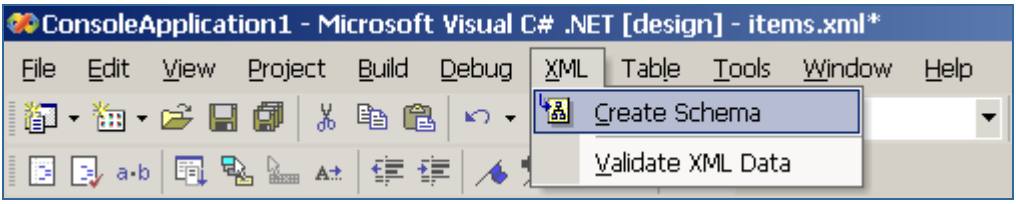
Когато XDR-съвместим парсер срещне пространство от имена, започващо с **x-schema**, той изтегля схемата от зададения URL адрес и извършва необходимата валидация.

Редакторът за схеми на VS.NET

VS.NET има силна поддръжка на XML и мощен редактор за XSD схеми. Нека разгледаме каква функционалност предоставя той.

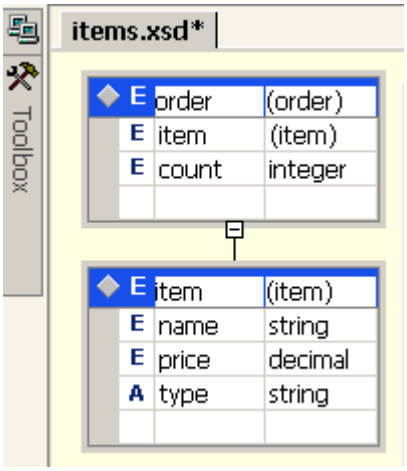
Поддръжка за XSD

Visual Studio .NET притежава вградена поддръжка за работа с XSD схеми. VS.NET позволява създаването на XSD схема по структурата на зададен XML документ (доколкото това е възможно). За целта при избран XML документ може да отидем в менюто XML и да изберем командата **Create Schema**. От същото меню може и да валидираме документ по вече създадена XSD схема с командата **Validate XML Data**.



Редактор за XSD схеми

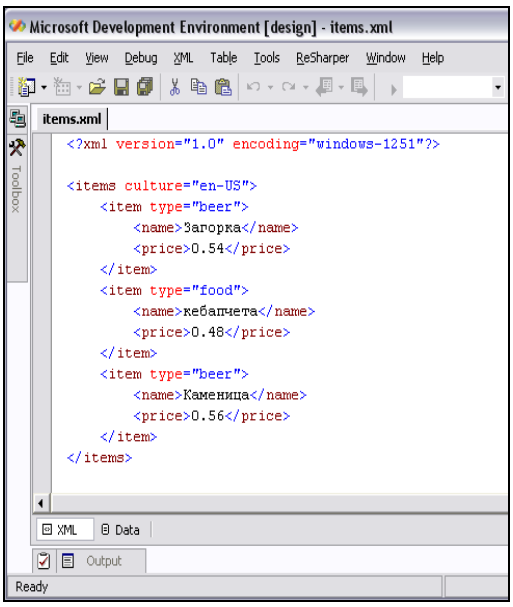
Visual Studio.NET разполага с вграден редактор за XSD схеми. Наред с възможността за промяна на XSD схемата в текстов режим, редакторът позволява и визуален режим на работа. Визуалният режим не е функционално ограничен – с него могат да се създават, променят и изтриват елементи, атрибути и типове, както и в текстовия режим на работа.

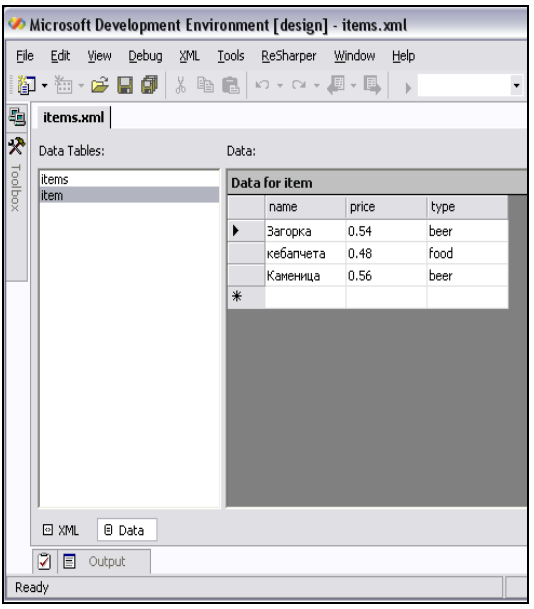


Създаване на XSD схема в VS.NET – пример

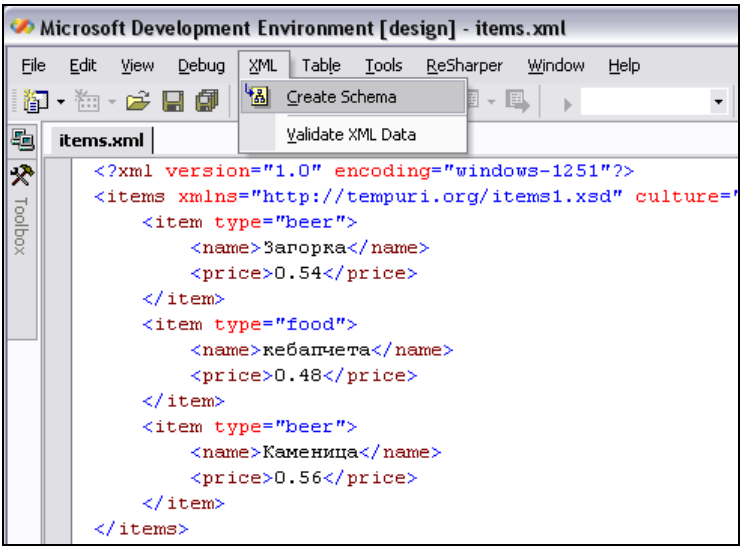
С настоящия пример ще демонстрираме как по визуален начин могат да се създават XSD схеми по наличен XML документ (в този случай **`items.xml`**). Ето и стъпките, които трябва да изпълним, за да получим желаната схема:

1. Стартираме VS.NET и отваряме XML документа **`items.xml`**. Файлът може да бъде разглеждан и редактиран освен като XML текст и като таблица с данни (смяната на двата режима правим от специален таб "**`XML/Data`**" в долната част на XSD редактора).

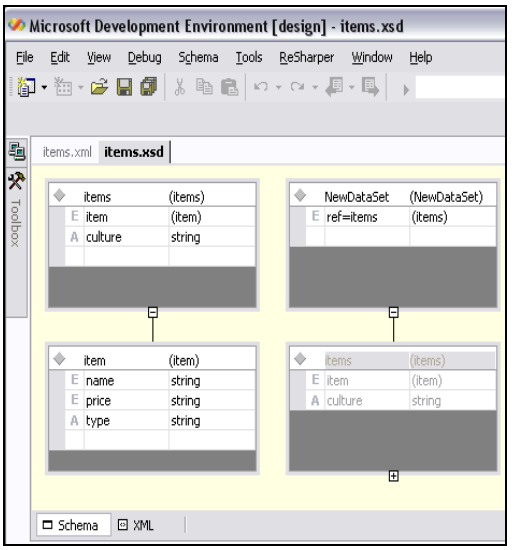


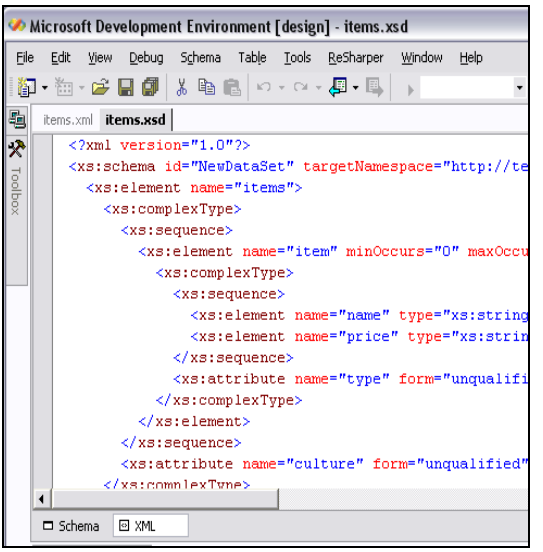


2. От менюто "**`XML`**" избираме "**`Create Schema`**" и VS.NET създава XSD схема на базата на структурата на отворения XML документ.



3. Схемата, генерирана от VS.NET, можем да редактираме както като чист XML текст, така и чрез визуалния редактор за схеми (смяната на двата режима отново става от специален таб "**`Schema/XML`**" в долната част на XSD редактора).



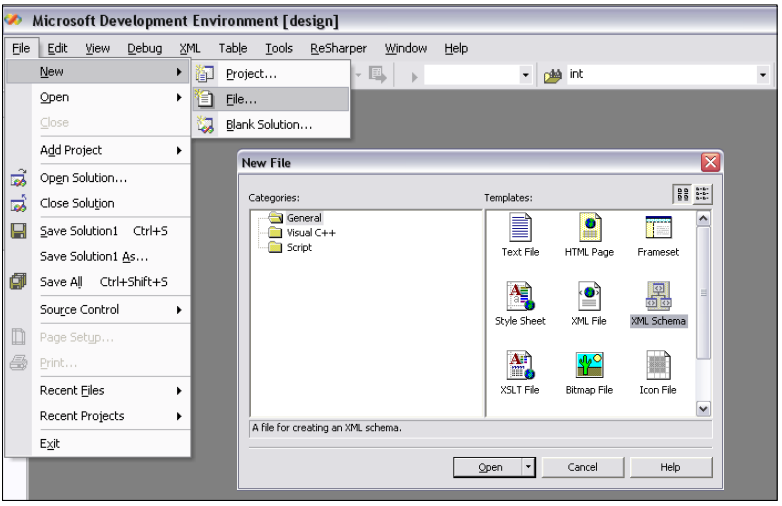


Редактиране на XSD схеми в VS.NET – пример

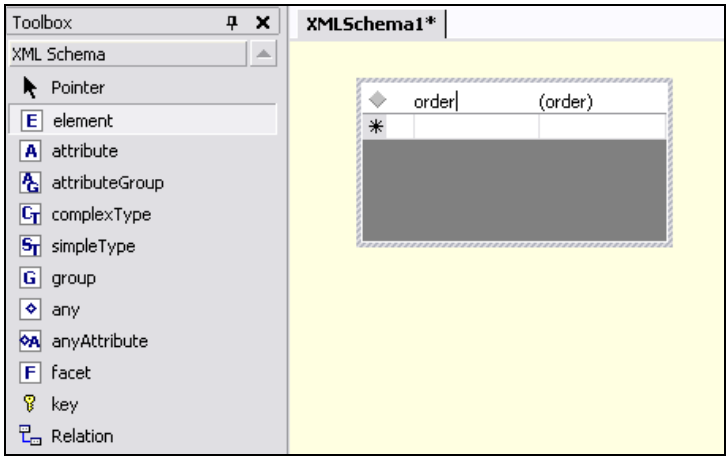
Сега ще демонстрираме как по визуален начин можем да редактираме XSD схеми във VS.NET. Ще създадем нова XSD схема, в която се дефинират три елемента – **order**, **items** и **item**. За всеки от тях визуално ще дефинираме необходимите елементи и атрибути и техните типове. Накрая отново във визуален режим на работа ще вградим елемента **item** в **items** и на свои ред **items** в **order**. Така със средствата на Visual Studio .NET ще създадем една примерна йерархична XSD схема, която можем да използваме за валидация на поръчки.

Нека проследим примера стъпка по стъпка:

1. Избираме **File | New | File… | XML Schema** от менюто на VS.NET:

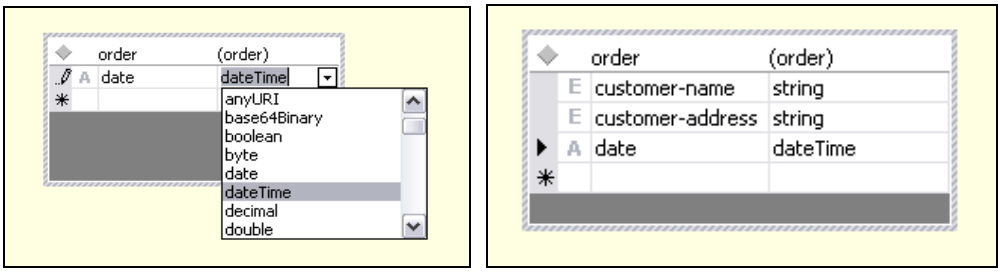


2. От Toolbox прозореца на VS.NET довличаме в схемата една контрола с име **"element"** и задаваме име на елемента **"order"**.

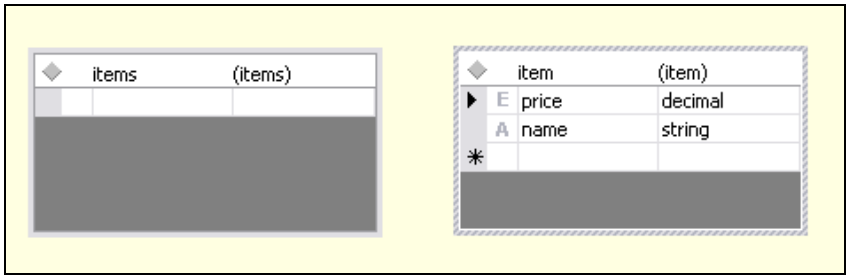


3. Добавяме в елемента **"order"** атрибут **"date"** от тип **"dateTime"**.

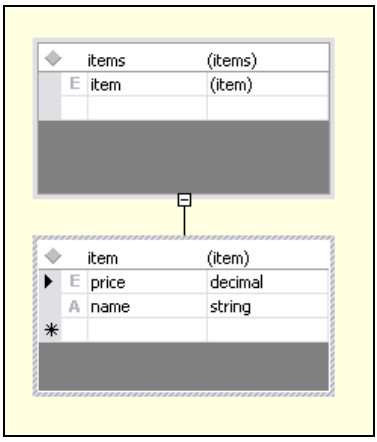
4. Добавяме в елемента **"order"** елементи **"customer-name"** и **"customer-address"** от тип **"string"**.



5. От Toolbox на VS.NET довличаме в схемата два нови елемента и им задаваме имена съответно **"items"** и **"item"**. В елемента **"item"** добавяме атрибут **"name"** от тип **"string"** и елемент **"price"** от тип **"decimal"**.

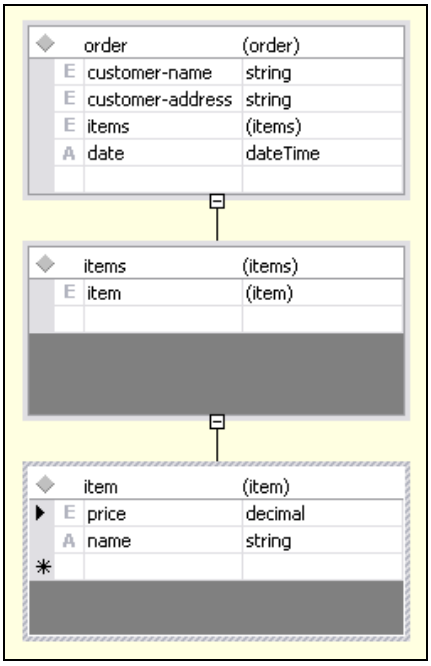


6. Довличаме елемента **"item"** върху елемента **"items"** и VS.NET автоматично влага в елемента **"items"** елемента **"item"**.



7. Довличаме елемента **"item"** върху елемента **"items"** и VS.NET автоматично влага в елемента **"items"** елемента **"item"**.

8. Накрая довличаме елемента **"items"** върху елемента **"order"**. VS.NET автоматично влага в елемента **"order"** елемента **"items"**. В крайна сметка се получава така, че в елемента **"order"** е вложен елемента **"items"**, а в него е вложен елементът **"item"**.



XML парсери

Терминът **парсване** на един език се описва като процес на взимането на фрагмент код, описан в синтаксиса на този език, и разбиването му на отделни компоненти, дефинирани от езиковите правила. Понякога на

български език се използва терминът "синтактичен анализ", макар че той не винаги е точен превод на оригиналния английския термин **parse**.

XML парсерите са библиотеки, които четат XML документи, извличат от тях таговете и съдържаната в тях информация и ги предоставят за обработка на програмиста. Те предоставят и функционалност за построяване на нови и промяна на вече създадени XML документи.

XML парсери – приложение

XML парсерите предоставят стандартизиран интерфейс за някои основни операции, свързани с обработката на XML данни:

- извличане на данни от XML документи
- построяване на нови XML документи
- промяна на съществуващи XML документи
- валидация на XML документи по зададена схема

XML парсери – видове

XML парсерите могат да се класифицират по различни критерии. От една страна те се делят на валидиращи (нуждаят се от DTD или XSD схема, по която да валидират документите) и невалидиращи (изискват единствено добре дефинирани документи, които да обработват). По начина на работа се разграничават дървовидно-ориентирани (DOM, Document Object Model) и поточно-ориентирани (SAX, Simple API for XML Processing) парсери. Ще разгледаме накратко особеностите на последните два модела.

DOM

Ще започнем с DOM стандарта и ще разгледаме обектния модел, който той дефинира.

Какво представлява DOM?

Документният обектен модел DOM (Document Object Model) дефинира платформено и езиково-независим програмен интерфейс за достъп и манипулиране на съдържанието и структурата на XML документите като дървовидни структури в паметта. XML документния обектен модел е базиран на W3C DOM спецификацията и е утвърден световен стандарт. Той не е технология, специфична за .NET.

Документният обектен модел представя един XML документ като дървовидна йерархия от възли. DOM стандартът дефинира следните типове възли: **`Document`**, **`Element`**, **`DocumentFragment`**, **`DocumentType`**, **`Attr`**, **`Text`**, **`EntityReference`**, **`ProcessingInstruction`**, **`Comment`**, **`CDATASection`**, **`Entity`** и **`Notation`**. Някои от тези типове могат да имат наследници, като за всеки възел те са определени в DOM спецификацията. Документният обектен

модел определя също типовете **`NodeList`** за обработка на колекции и **`NamedNodeMap`** за речникови обекти от тип ключ-стойност.

DOM спецификацията описва интерфейси, а не действителни класове и обекти и затова за работа с нея ни е необходима конкретна имплементация (DOM парсер).

DOM приложения

DOM не е универсално решение, подходящо за всички случаи на обработка на XML документи. DOM обектната йерархия съхранява референции между различните възли в един документ. За целта целият XML документ трябва да е прочетен и парснат преди да бъде подаден на DOM приложението. При обработката на обемисти XML документи това може да се окаже сериозен проблем, защото е необходимо съхранението на целия документ в паметта. Въпреки това документният обектен модел е отлично решение в много ситуации. При нужда от произволен достъп до различни части от XML документа по различно време или за приложения, които променят структурата на XML документа "в движение", DOM предоставя отлична функционалност.

SAX

Сега нека сега разгледаме и SAX стандарта за обработка на XML документи и изясним кога е подходящо да се използва.

Какво представлява SAX?

SAX (Simple API for XML Processing) е базиран на събития, програмен интерфейс, който чете XML документи последователно като поток и позволява анализиране на съдържанието им.

Обработката на XML документи, базирана на събития, следи за наличието на ограничен брой събития, като срещане на отварящи и затварящи тагове на елементи, character data, коментари, инструкции за обработка и др. В процеса на прочитане на един документ SAX парсерът изпраща информация за документа в реално време чрез обратни извиквания. Всеки път, когато парсерът срещне отварящ или затварящ таг, character data или друго събитие, той известява за това програмата, която го използва.

SAX приложения

При SAX базираните приложения XML документът се предоставя за обработка на програмата фрагмент по фрагмент от началото до края. SAX приложението може да съхранява интересуващите го части, докато целият документ бъде прочетен, или може да обработва информацията в момента на получаването й. Не е нужно обработката на вече прочетени елементи да чака прочитането на целия документ и още по-важно – не е нужно целият документ да се съхранява в паметта, за да е възможна работата с него. Тези характеристики правят SAX модела много удобен за обработка на обемисти XML документи, които не могат да бъдат заредени в паметта.

XML и .NET Framework

До момента направихме преглед на XML стандарта и по-важните технологии, свързани с него. Вече имаме стабилна основа, за да продължим с програмните средства, които .NET Framework предоставя за обработка на XML документи.

.NET притежава вградена XML поддръжка

За разлика от много програмни езици и платформи, който осигуряват средства за работа с XML под формата на добавки към основната функционалност, .NET Framework е проектиран от самото начало с идеята за силно интегрирана XML поддръжка.

Имплементациите на основните XML технологии се съдържат в асемблито **`System.Xml`**, където са дефинирани следните главни пространства от имена:

- **`System.Xml`** – осигурява основните входно-изходни операции с XML (**`XmlReader`** и **`XmlWriter`**), DOM поддръжка (**`XmlNode`** и наследниците му) и други XML помощни класове.
- **`System.Xml.Schema`** – осигурява поддръжка на валидация на XML съдържание чрез XML Schema (**`XmlSchemaObject`** и наследниците му).
- **`System.Xml.XPath`** – реализира функционалност за XPath търсене на информация и навигация в XML документ (класовете **`XPathDocument`**, **`XPathNavigator`** и **`XPathExpression`**).
- **`System.Xml.Xsl`** – предоставя възможност за XSL трансформации на XML документи (**`XslTransform`**).
- **`System.Xml.Serialization`** – осигурява сериализация до XML и SOAP (**`XmlSerializer`**).

.NET и DOM моделът

.NET Framework предоставя пълен набор от класове, които зареждат и редактират XML документи според W3C DOM спецификацията (нива 1 и 2). Основният XML DOM клас в .NET Framework е **`XmlDocument`**. Силно свързан с него е неговият клас-наследник **`XMLDataDocument`**, който разширява **`XMLDocument`** и акцентира върху съхраняването и извличането на структурирани таблични данни в XML.

При работа с XML DOM модела XML документът първо се зарежда целият като дърво в паметта и едва тогава се обработва. XML DOM предоставя средства за навигация и редактиране на XML документа и поддържа XPath заявки и XSL трансформации (ще ги разгледаме малко по-нататък).

Парсване на XML документ с DOM – пример

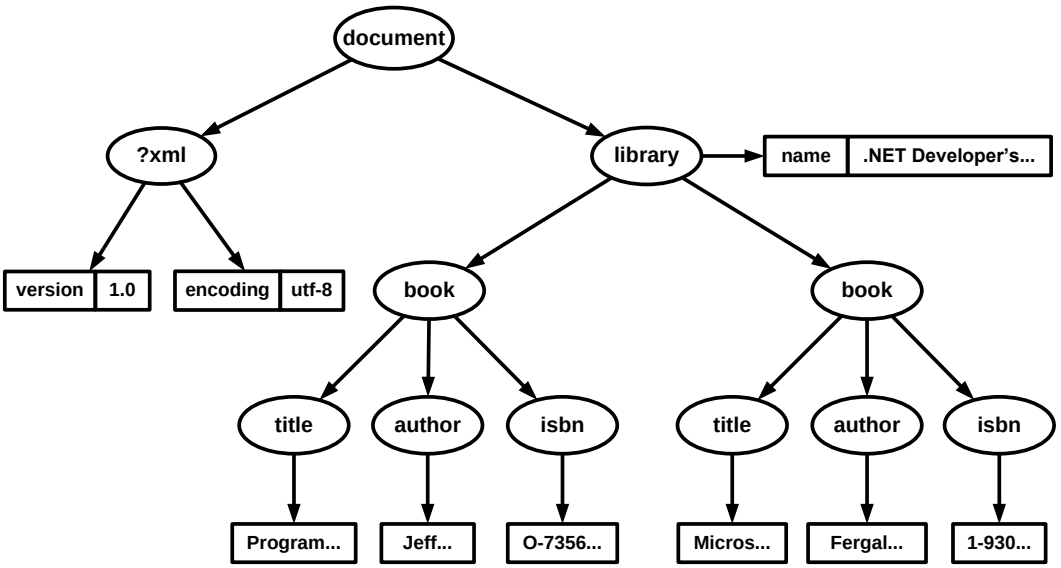
Преди да навлезем в детайлите на DOM парсера в .NET Framework, ще разгледаме кратък пример, който илюстрира използването му за парсване на XML документ, обхождане на полученото DOM дърво и извличане на информация от него.

За целта ни е необходим работен XML документ:

library.xml

```
<?xml version="1.0"?>
<library name=".NET Developer's Library"
   <book>
      <title>Programming Microsoft .NET</title>
      <author>Jeff Prosise</author>
      <isbn>0-7356-1376-1</isbn>
   </book>
   <book>
      <title>Microsoft .NET for Programmers</title>
      <author>Fergal Grimes</author>
      <isbn>1-930110-19-7</isbn>
   </book>
</library>
```

Този документ се представя в паметта като DOM дърво по следния начин:



Нашият пример има за цел да извлече книгите от файла **library.xml** и да отпечата информация за тях – заглавие, автор и ISBN. Ще го изградим стъпка по стъпка:

1. Създаваме нов проект във VS.NET.

2. Първото, което е необходимо да направим, е да заредим XML файла **library.xml**, за да можем в последствие да го подложим на обработка. След зареждането на документа отпечатваме съдържанието му, за да се уверим, че зареждането е успешно:

```
XmlDocument doc = new XmlDocument();
doc.Load("library.xml");
Console.WriteLine("Loaded XML document:");
Console.WriteLine(doc.OuterXml);
Console.WriteLine();
```

3. Извличаме документния елемент на XML файла и отпечатваме името му на конзолния изход:

```
XmlNode rootNode = doc.DocumentElement;
Console.WriteLine("Root node: {0}", rootNode.Name);
```

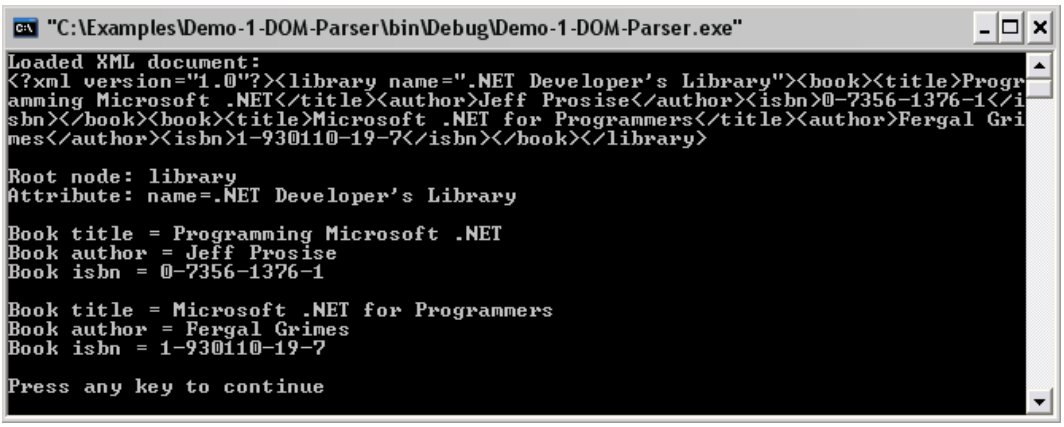
4. Обхождаме и отпечатваме атрибутите на документния елемент (в този случай имаме един единствен атрибут **name**):

```
foreach (XmlAttribute atr in rootNode.Attributes)
{
   Console.WriteLine("Attribute: {0}={1}",
      atr.Name, atr.Value);
}
```

5. Обхождаме всички елементи-деца на документния елемент. Всеки от тях описва една книга (елемент **book**). За всяка книга отпечатваме заглавието, автора и isbn номера, като ги извличаме от съответните им елементи, наследници на елемента **book**:

```
foreach (XmlNode node in rootNode.ChildNodes)
{
   Console.WriteLine("Book title = {0}",
      node["title"].InnerText);
   Console.WriteLine("Book author = {0}",
      node["author"].InnerText);
   Console.WriteLine("Book isbn = {0}",
      node["isbn"].InnerText);
   Console.WriteLine();
}
```

6. Това е всичко. Ето го резултатът, който получаваме на конзолния изход след компилацията и изпълнението на проекта:



В примера използвахме класовете **`XmlDocument`**, **`XmlNode`** и **`XmlAttribute`**, от пространството **`System.Xml`**. Нека разгледаме за какво служат и как се използват тези класове.

Класовете за работа с DOM

Работата с DOM в .NET Framework се осъществява с помощта на следните по-важни класове:

- **`XmlNode`** – абстрактен базов клас за всички възли в едно DOM дърво.
- **`XmlDocument`** – съответства на корена на DOM дърво, обикновено съдържа два наследника: заглавна част (пролог) и документния елемент на XML документа.
- **`XmlElement`** – представя XML елемент.
- **`XmlAttribute`** – представя атрибут на XML елемент (двойка име-стойност).
- **`XmlAttributeCollection`** – списък от XML атрибути.
- **`XmlNodeList`** – списък от възли в DOM дърво.

Класът XmlNode

Да разгледаме най-важния клас в обектния модел на .NET за работа с XML – класът **`XmlNode`**.

Основополагащ при работата с DOM

Класът **`XmlNode`** е абстрактният клас, който представя възел в един XML документ. Той имплементира стандартизирания от W3C документен обектен модел (нива 1 и 2) и е ключът към работата с DOM в .NET Framework. Възли в един документ могат да бъдат елементи, атрибути, DOCTYPE декларации, коментари и дори целият XML документ.

XmlNode е базов клас за различните DOM типове възли

Класът **XmlNode** представя базов възел и е класът, наследяван от всички специфични DOM възли (**XmlDocument**, **XmlElement**, **XmlAttribute** и т.н.). Неговите свойства осигуряват достъп до вътрешните стойности на всеки възел: пространството от имена на възела, тип на възела, възел-родител, възел-наследник, съседни възли и др.

XmlNode позволява навигация в DOM дървото

Класът **XmlNode** предоставя набор от средства за навигация чрез своите свойства:

- **ParentNode** – връща възела-родител (или **null** ако няма).
- **PreviousSibling** / **NextSibling** – връща левия / десния съсед на текущия възел.
- **FirstChild** / **LastChild** – връща първия / последния наследник на текущия възел.
- **Item** (индексатор в C#) – връща наследник на текущия възел по името му.

XmlNode позволява работа с текущия възел в DOM дървото

- **Name** – връща името на възела (име на елемент, атрибут, ...).
- **Value** – връща стойността на възела.

Стойността на свойството `Value` в голяма степен зависи от типа на конкретно разглеждания възел. За възел от тип атрибут това свойство наистина връща стойността му, но за възел от тип елемент например, `Value` връща нулева референция. Стойността на елементите се достъпва през свойствата `InnerText` и `InnerXml`. За пълен списък на връщаните от `Value` стойности за различните DOM възли потърсете в MSDN.

- **Attributes** – връща списък от атрибутите на възела (като **XmlAttributeCollection**).
- **HasChildNodes** – връща булева стойност дали има възелът има наследници.
- **InnerXml**, **OuterXml** – връща частта от XML документа, която описва съдържанието на възела съответно без и с него самия.
- **InnerText** – връща конкатенация от стойностите на възела и наследниците му рекурсивно.
- **NodeType** – връща типа на възела (вж. изброения тип **XmlNodeType** в MSDN).

XmlNode позволява промяна на текущия възел

- **AppendChild(…)** / **PrependChild(…)** – добавя нов наследник след / преди всички други наследници на текущия възел.
- **InsertBefore(…)** / **InsertAfter(…)** – вмъква нов наследник преди / след указан наследник.
- **RemoveChild(…)** / **ReplaceChild(…)** – премахва / заменя указания наследник.
- **RemoveAll()** – изтрива всички наследници на текущия възел (атрибути, елементи, ...).
- **Value**, **InnerText**, **InnerXml** – променя стойността / текста / XML текста на възела.

Класът XmlDocument

Класът **XmlDocument** съдържа един XML документ във вид на DOM дърво според W3C спецификацията за документния обектен модел. Документът е представен като дърво от възли, които съхраняват елементите, атрибутите и техните стойности и съдържат информация за родител, наследник и съседни възли.

Да разгледаме неговите основни свойства, методи и събития

- **Load(…)**, **LoadXml(…)**, **Save(…)** – позволяват зареждане и съхранение на XML документи от и във файл, поток или символен низ
- **DocumentElement** – извлича документния елемент на XML документа.
- **PreserveWhitespace** – указва дали празното пространство да бъде запазено при зареждане / записване на документа.
- **CreateElement(…)**, **CreateAttribute(…)**, **CreateTextNode(…)** – създава нов XML елемент, атрибут или стойност на елемент.
- **NodeChanged**, **NodeInserted**, **NodeRemoved** – събития за следене за промени в документа.

Промяна на XML документ с DOM – пример

Ще разгледаме кратък пример, който демонстрира приложението на DOM парсера на .NET Framework за промяна на съдържанието на XML документ.

За работен XML документ ще използваме **items.xml**:

items.xml

```
<?xml version="1.0" encoding="windows-1251"?>
<items>
  <item type="beer">
```

```
    <name>Загорка</name>
    <price>0.54</price>
  </item>
  <item type="food">
    <name>кебапчета</name>
    <price>0.48</price>
  </item>
  <item type="beer">
    <name>Каменица</name>
    <price>0.56</price>
  </item>
</items>
```

Поставената задача е да удвоим цените на бирата в този XML документ, но същевременно да запазим непроменени цените на останалите стоки в списъка. За целта ще е необходимо да прочетем целия XML документ в паметта и да анализираме стоките една по една. При срещане на елемент, който идентифицираме като бира, удвояваме цената му, а в противен случай не предприемаме никакво действие.

Нека сега разгледаме стъпките за изграждане на приложението:

1. Стартираме VS.NET и създаваме нов проект – конзолно приложение.
2. Зареждаме работния XML документ **items.xml** в паметта, за да го подготвим за предстоящата манипулация:

```
XmlDocument doc = new XmlDocument();
doc.Load("items.xml");
```

3. Естеството на този пример ни задължава да работим с десетични числа. В XML документа те са форматирани с десетична точка, но винаги съществува вероятност текущата активна култура на компютъра, където изпълняваме програмата, да е различна и да форматира числата с десетична запетая (например българската култура). За да се подсигурим, че парсването на числата ще протече безпроблемно и няма да предизвика изключение от тип **FormatException**, най-правилно е да използваме специалната културно-необвързана култура, достъпна през свойството **CultureInfo.InvariantCulture**.

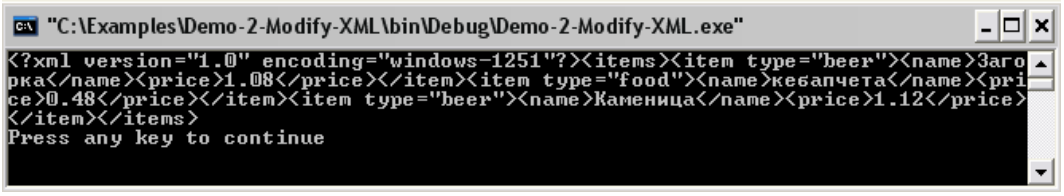
4. Обхождаме наследниците **item** на документния елемент **items** и за всеки от тях, чийто атрибут **type** има стойност **"beer"**, прочитаме стойността на наследника му **price**. Дотук обаче имаме стойността на елемента **price** единствено като низ. Необходимо е да парснем низа към десетично число и именно тук използваме **CultureInfo.InvariantCulture**. Вече разполагаме с десетично число, което удвояваме и записваме на мястото на старата стойност на елемента **price** (отново е нужно да укажем културата, за да се предпазим от грешки). Ето как изглежда кода, който извършва манипулацията:

```
foreach (XmlNode node in doc.DocumentElement)
{
  if (node.Attributes["type"].Value == "beer")
  {
    string currentPriceStr =
      node["price"].InnerText;
    decimal currentPrice = Decimal.Parse(
      currentPriceStr, CultureInfo.InvariantCulture);
    decimal newPrice = currentPrice * 2;
    node["price"].InnerText =
      newPrice.ToString(CultureInfo.InvariantCulture);
  }
}
```

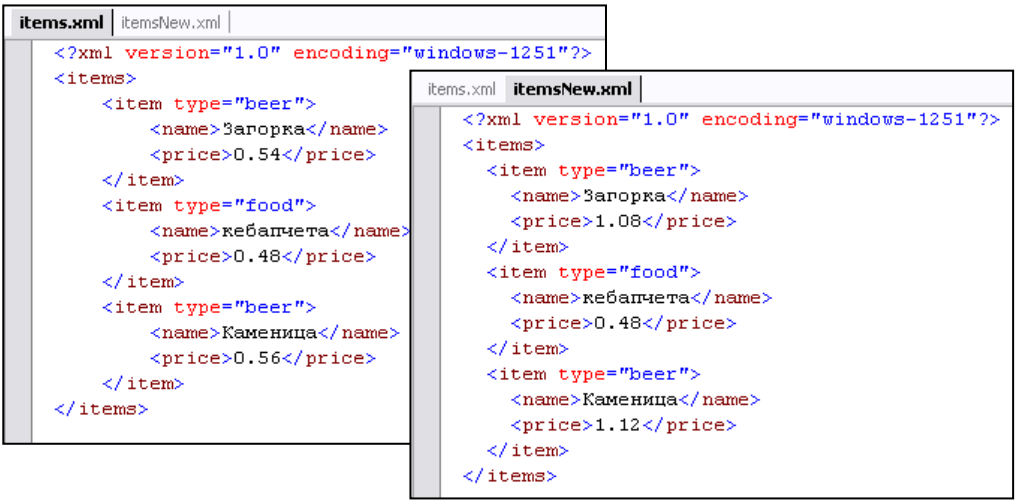
5. Сега остава единствено да отпечатаме XML документа, за да се уверим, че промените действително са налице и след това да запазим промените в нов файл **itemsNew.xml**:

```
Console.WriteLine(doc.OuterXml);

doc.Save("itemsNew.xml");
```

6. Това е всичко. Ето резултата, който получаваме на конзолния изход след компилацията и изпълнението на проекта:



Можем да се уверим, че условието на задачата е изпълнено, като сравним двата XML документа **items.xml** и **itemsNew.xml**:



Построяване на XML документ с DOM – пример

За да илюстрираме по-пълно работата с DOM, ще разгледаме още един пример. Да си поставим за задача построяването на следния XML документ:

order.xml

```
<order>
  <item ammount="4">бира</item>
  <item ammount="2">картофки</item>
  <item ammount="6">кебапчета</item>
</order>
```

За целта трябва да създадем **`XmlDocument`**, да създадем и добавим документен елемент като негов наследник, след което да добавим към документния елемент още 3 елемента, като им зададем подходящо съдържание и им добавим по един атрибут за количество.

Ето примерна програма на C#, която реализира описаните стъпки:

```
using System.Xml;

class CreateXmlDemo
{
   static void AppendItem(XmlDocument aXmlDoc, XmlElement
      aXmlElement, string aItemName, int aAmmount)
   {
      XmlElement itemElement = aXmlDoc.CreateElement("item");
      itemElement.InnerText = aItemName;
      XmlAttribute ammountAttr =
         aXmlDoc.CreateAttribute("ammount");
      ammountAttr.Value = aAmmount.ToString();
      itemElement.Attributes.Append(ammountAttr);
      aXmlElement.AppendChild(itemElement);
   }

   static void Main()
   {
      XmlDocument xmlDoc = new XmlDocument();
      XmlElement docElement = xmlDoc.CreateElement("order");
      xmlDoc.AppendChild(docElement);
      AppendItem(xmlDoc, docElement, "бира", 4);
      AppendItem(xmlDoc, docElement, "картофки", 2);
      AppendItem(xmlDoc, docElement, "кебапчета", 6);
      xmlDoc.Save("order.xml");
   }
}
```

SAX парсери и XmlReader

В .NET Framework няма чиста имплементация на SAX парсер. Класът **`XmlReader`** има функционалност подобна на тази, предоставяна от класическите SAX парсери, но между тях има и определени разлики, които ще разгледаме по-подробно.

Класът XmlReader

`XmlReader` е абстрактен клас, който осигурява поточно-ориентиран еднопосочен достъп до XML данни само за четене. **`XmlReader`** е базиран на събития, както и SAX парсерите, но за разлика от тях е представител на pull модела, докато SAX парсерите по идея са push-ориентирани (двете понятия ще обясним малко по-надолу). Едно събитие указва начало или край на възел в процеса на прочитането му от потока от данни. В **`XmlReader`** информацията за настъпило събитие е достъпна през свойствата на класа, след като е извикан неговият **`Read()`** метод.

Разлика между pull и push парсер моделите

Съществуват два модела на работа на XML парсерите, обработващи поточно документи – push модел и pull модел.

Push парсер

Начинът на работа на push парсерите се характеризира с пряк контрол върху процеса на парсване, като настъпващите събития се предават **без изчакване** към клиентското приложение. Обикновено един push парсер изисква да се регистрира функция за обратно изискване (callback функция), която да обработва всяко събитие при настъпването му. Клиентското приложение не може да контролира парсването и трябва да съхранява информация за състоянието на парсера във всеки един момент, за да могат callback функциите да се изпълняват в правилен контекст (например трябва да помни колко дълбоко в XML дървото се намира в момента).

Pull парсер

При pull парсерите клиентското приложение упражнява активен контрол върху парсера. То изпълнява цикъл по събитията, идващи от парсера, като **изрично** извлича всяко следващо събитие. Приложението може да дефинира методи за обработката на специфични събития и изцяло да пропуска обработката на други, които не го интересуват. Това осигурява по-голяма ефикасност в сравнение с push модела, където всички данни задължително минават през клиентското приложение, защото само то може да прецени кои данни представляват интерес за него и кои – не.

XmlReader – основни методи и свойства

Нека сега направим преглед на най-важните методи и свойства на класа **XmlReader**:

- **Read()** – прочита следващия възел от XML документа или връща **false**, ако няма следващ
- **NodeType** – връща типа на прочетения възел
- **Name** – връща името на прочетения възел (име на елемент, на атрибут, ...)
- **HasValue** – връща дали възелът има стойност
- **Value** – връща стойността на възела
- **ReadElementString()** – прочита стойността (текста) от последния прочетен елемент
- **AttributeCount**, **GetAttribute(…)** – за извличане на атрибутите на XML елемент

Класът XmlReader – начин на употреба

XmlReader е абстрактен клас и осигурява само най-съществената функционалност за четене на XML документи. За работа с него се използват неговите наследници:

- **XmlTextReader** – за четене от файл или поток
- **XmlNodeReader** – за четене от възел в DOM дърво
- **XmlValidatingReader** – за валидация по XSD, DTD или XDR схема при четене от друг **XmlReader**

XmlReader – пример

Ще разгледаме кратък пример, който илюстрира работата с **XmlReader** за извличане на информация от XML документ.

Ще използваме следния работен документ:

library.xml

```
<?xml version="1.0"?>
<library name=".NET Developer's Library">
  <book>
    <title>Programming Microsoft .NET</title>
    <author>Jeff Prosise</author>
    <isbn>0-7356-1376-1</isbn>
  </book>
  <book>
```

```
    <title>Microsoft .NET for Programmers</title>
    <author>Fergal Grimes</author>
    <isbn>1-930110-19-7</isbn>
  </book>
</library>
```

Целта на примера е да извлечем всички заглавия на книги, които се съдържат в XML документа и после да извлечем имената на всички елементи от документа.

Нека разгледаме необходимите стъпки:

1. Стартираме VS.NET и създаваме нов проект.
2. Отпечатваме на конзолния изход съобщение, че ще извличаме заглавията на книги от документа и инициализираме XML четеца:

```
Console.WriteLine("Book titles in the library:");
XmlTextReader reader = new XmlTextReader("library.xml");
```

3. Започваме да четем възлите един по един (при pull-базирания парсер **`XmlReader`** това означава, че изпълняваме цикъл по събитията, като с **`Read()`** извличаме всяко следващо събитие (т.е. възел). За всеки прочетен възел проверяваме дали е от тип елемент и дали името му съответства на търсените от нас заглавия на книги (**`"title"`**). В случай че възелът отговаря на тези условия, отпечатваме текстовата му стойност на конзолния изход:

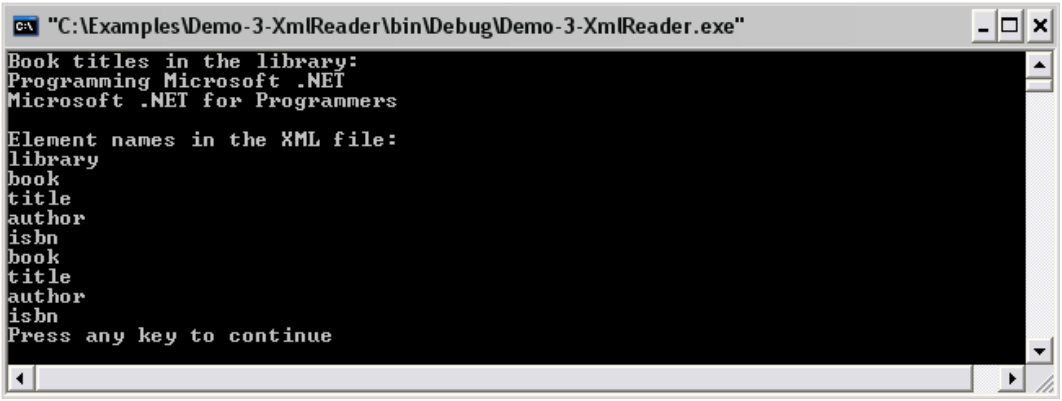
```
while (reader.Read())
{
  if ((reader.NodeType == XmlNodeType.Element) &&
     (reader.Name == "title"))
  {
    Console.WriteLine(reader.ReadElementString());
  }
}
```

4. Дотук извлякохме всички заглавия на книги и ги отпечатахме. Сега остава да изпълним втората част от задачата – да отпечатаме имената на всички елементи от XML документа. Тъй като **`XmlReader`** е еднопосочен поточно-ориентиран парсер, необходимо е отново да го инициализираме преди да можем да го използваме. Извличането на имената на всички елементи е аналогично на описаното по-горе извличане на заглавията на книгите – отново влизаме в цикъл по събитията, като този път проверката се състои само в това да установи дали възлите са от тип елемент. При положение, че условието е изпълнено, отпечатваме името на текущия елемент на конзолния изход:

```
Console.WriteLine("\nElement names in the XML file:");
```

```
reader = new XmlTextReader("library.xml");
while (reader.Read())
{
   if (reader.NodeType == XmlNodeType.Element)
   {
      Console.WriteLine(reader.Name);
   }
}
```

5. Ето резултата, който получаваме на конзолния изход след като компилираме и изпълним програмата:



Кога да използваме DOM и SAX?

Моделът за обработка на XML документи DOM (**`XmlDocument`**) е подходящ, когато:

- обработваме малки по обем документи
- нуждаем се от гъвкавост при навигацията
- имаме нужда от пряк достъп до отделните възли на документа
- желаем да променяме документа

Моделът за обработка на XML документи SAX (**`XmlReader`**) е подходящ когато:

- обработваме големи по обем документи
- скоростта на обработка е важна
- не е необходимо да променяме възлите на документа

Класът XmlWriter

Класът **`XmlWriter`** осигурява бърз, еднопосочен, поточно-ориентиран способ за записване на XML данни във файлове и потоци. **`XmlWriter`** дефинира специални методи за записване различните съставни части на

един XML документ (елементи, атрибути, инструкции за обработка, коментари, и др.). XML писачът гарантира, че данните, които излизат от него, се съобразяват с W3C XML 1.0 стандарта и W3C спецификациите за пространства от имена.

XmlWriter генерира добре дефинирани документи, но само при положение, че потребителят подава коректна информация в процеса на създаване на нов XML документ.

XmlWriter не прави проверка за невалидни символи в имената на елементите и атрибутите. XML писачът не дава гаранции, че евентуална употреба на Unicode символи от страна на потребителя отговаря на текущата кодова таблица. В резултат на това символите без съответствие в кодовата таблица не се ескейпват и това може да доведе до некоректен изходен документ.

XmlWriter не проверява за дублирани имена на атрибути, нито валидира идентификаторите, задавани от потребителя при създаване на **DOCTYPE** възел (например **SYSTEM** идентификатора).

XmlWriter не осигурява вградена валидация по схема или DTD декларация.

XmlWriter – основни методи

- **WriteStartDocument()** – добавя стандартна XML 1.0 пролог декларация в началото на документа (**<?xml** …).
- **WriteStartElement(…)** – добавя отварящ таг за зададения елементен възел.
- **WriteEndElement()** – затваря най-вътрешния отворен елемент (използва кратък затварящ таг (**/>**), където е възможно).
- **WriteAttributeString(…)** – добавя атрибут в текущия елемент (методът добавя автоматично отварящи и затварящи кавички).
- **WriteElementString(…)** – добавя елемент по зададено име и текстова стойност.
- **WriteEndDocument()** – затваря всички отворени тагове и изпразва вътрешните буфери (чрез **Flush()**).

Работа с XmlWriter

XmlWriter е абстрактен клас, въпреки че някои от методите му имат конкретна имплементация. За работа с **XmlWriter** в .NET Framework се използва единственият му наследник – **XmlTextWriter**.

XmlTextWriter поддържа запис на XML данни в поток, файл или **TextWriter**. В конструктора му се задава необходимата кодираща схема

или се използва UTF-8 кодиране по подразбиране, ако не е определена схема. Класът осигурява стандартна имплементация на методите и свойствата на абстрактния **XmlWriter**, като към тях добавя и някои свои свойства:

- **Formatting** – избираме **Formatting.None**, ако XML данните не изискват отместване и **Formatting.Indented** в случай, че търсим подобрена четимост на документа (съдържанието на XML е идентично и при двата вида форматиране).
- **Indentation** – при избрана опция **Formatting.Indented**, **Indentation** определя броя символи, с които отместваме всяко следващо markup ниво.
- **IndentChar** – при избрана опция **Formatting.Indented**, **IndentChar** определя символа, който ще използваме за отместване на всяко следващо markup ниво. За да осигурим валидността на XML документа, трябва да използваме валидни XML символи за празно пространство.
- **Namespaces** – определя дали **XmlTextWriter** поддържа W3C XML пространства от имена.
- **QuoteChar** – определя какви кавички ще се използват при дефиниране на стойности на атрибути. **QuoteChar** може да бъде единична или двойна кавичка, всеки друг символ ще предизвика хвърляне на изключение от тип **ArgumentException**.

XmlWriter – пример

Ще разгледаме един кратък пример, който илюстрира работата с **XmlWriter** за създаване на нов XML документ. Целта на демонстрацията е да запишем информацията за няколко книги (заглавие, автор и isbn) от малка домашна библиотека в XML документ.

Нека разгледаме необходимите стъпки:

1. Стартираме VS.NET и създаваме нов проект
2. Създаваме нов обект от клас **XmlTextWriter**, като в конструктора подаваме името на изходния XML файл и избраната от нас кодираща схема **windows-1251**:

```
XmlTextWriter writer = new XmlTextWriter("library.xml",
  Encoding.GetEncoding("windows-1251"));
```

3. За по-добра четимост на изходния документ указваме, че ще използваме една табулация като символ за отместване на всяко следващо markup ниво в документа:

```
writer.Formatting = Formatting.Indented;
writer.IndentChar = '\t';
```

```
writer.Indentation = 1;
```

4. Създаването на XML документа започва със записване на стандартен W3C XML 1.0 пролог в потока.

```
writer.WriteStartDocument();
```

5. Добавяме документен елемент с име **library** и дефинираме негов атрибут **name** със стойност **"My Library"**:

```
writer.WriteStartElement("library");
writer.WriteAttributeString("name", "My Library");
```

6. За записването на книгите използваме помощна функция **WriteBook(…)**, която приема като параметри референция към **XmlWriter** обект и информацията за всяка книга (заглавие, автор, isbn). **WriteBook(…)** записва нов елемент **book** и в негови директни наследници **title**, **author** и **isbn** записва подадената в параметрите на функцията информация. Накрая затваряме елемента **book**:

```
private static void WriteBook(XmlWriter aWriter,
   string aTitle, string aAuthor, string aIsbn)
{
   aWriter.WriteStartElement("book");
   aWriter.WriteElementString("title", aTitle);
   aWriter.WriteElementString("author", aAuthor);
   aWriter.WriteElementString("isbn", aIsbn);
   aWriter.WriteEndElement();
}
```

7. С помощта на функцията **WriteBook(…)** добавяме в XML документа информацията за няколко книги:

```
WriteBook(writer, "Code Complete",
   "Steve McConnell", "155-615-484-4");
WriteBook(writer, "Интернет програмиране с Java",
   "Светлин Наков", "954-775-305-3");
WriteBook(writer, "Writing Solid Code",
   "Steve Maguire", "155-615-551-4");
```

8. Сега остава единствено да затворим отворените тагове и да затворим **XmlTextWriter** обекта (така затваряме и потока, използван за писане във файла). Добре е тази операция да се извърши във **finally** клауза на **try-finally** блок (обхващащ записването на всички елементи в XML документа). По този начин сме сигурни, че и при непредвидени обстоятелства потокът не остава отворен:

```
    try
    {
       ...
       writer.WriteEndDocument();
    }
    finally
    {
       writer.Close();
    }
```

9. В крайна сметка получихме следната програма на C#:

```
using System;
using System.Xml;
using System.Text;

class XmlWriterDemo
{
   public static void Main()
   {
      XmlTextWriter writer = new XmlTextWriter("library.xml",
         Encoding.GetEncoding("windows-1251"));
      writer.Formatting = Formatting.Indented;
      writer.IndentChar = '\t';
      writer.Indentation = 1;
      try
      {
         writer.WriteStartDocument();
         writer.WriteStartElement("library");
         writer.WriteAttributeString("name", "My Library");
         WriteBook(writer, "Code Complete",
            "Steve McConnell", "155-615-484-4");
         WriteBook(writer, "Интернет програмиране с Java",
            "Светлин Наков", "954-775-305-3");
         WriteBook(writer, "Writing Solid Code",
            "Steve Maguire", "155-615-551-4");
         writer.WriteEndDocument();
      }
      finally
      {
         writer.Close();
      }
   }

   private static void WriteBook(XmlWriter aWriter,
      string aTitle, string aAuthor, string aIsbn)
   {
      aWriter.WriteStartElement("book");
      aWriter.WriteElementString("title", aTitle);
      aWriter.WriteElementString("author", aAuthor);
```

```
        aWriter.WriteElementString("isbn", aIsbn);
        aWriter.WriteEndElement();
    }

}
```

10. Компилираме и изпълняваме програмата. Ето как изглежда съдържанието на генерирания XML документ **library.xml**, получен след компилацията и изпълнението на програмата:

library.xml

```
<?xml version="1.0" encoding="windows-1251"?>
<library name="My Library">
    <book>
        <title>Code Complete</title>
        <author>Steve McConnell</author>
        <isbn>155-615-484-4</isbn>
    </book>
    <book>
        <title>Интернет програмиране с Java</title>
        <author>Светлин Наков</author>
        <isbn>954-775-305-3</isbn>
    </book>
    <book>
        <title>Writing Solid Code</title>
        <author>Steve Maguire</author>
        <isbn>155-615-551-4</isbn>
    </book>
</library>
```

Валидация на XML по схема

Схемите (XSD, DTD и XDR) описват правила и ограничения за съставяне на XML документи. Те указват позволените тагове, реда и начините на влагането им, контролират позволените атрибути и техните стойности. Валидацията на XML документ по дадена схема в .NET Framework се извършва с помощта на валидиращи парсери.

Класът XmlValidatingReader

XmlValidatingReader е имплементация на абстрактния клас **XmlReader**, която осигурява поддръжка на основните схеми за валидация – XSD, DTD и XDR. **XmlValidatingReader** може да се използва за валидация както на цели XML документи, така и на XML фрагменти (т. е. без документен елемент). Този клас не може да бъде инстанциран директно от файл или URL адрес. Валидиращите четци в .NET Framework винаги работят върху

вече съществуващ XML четец и затова **XmlValidationReader** имплементира вътрешно само малка част от функционалността на родителския клас **XmlReader**.

XmlValidatingReader – основни методи, свойства и събития

- **Schemas** – връща **XmlSchemaCollection** обект, който съдържа колекция от предварително заредени XDR и XSD схеми. Предварителното зареждане ускорява процеса на валидация - схемите се кешират и няма нужда да се зареждат всеки път.

Предварителното зареждане и кеширане на схемите в XmlValidatingReader.Schemas е възможно за XSD и XDR схеми, но не и за DTD декларации.

- **SchemaType** (приложим само за XSD схеми) – връща съответния схема-обект за текущия възел на XML четеца в основата на **XmlValidatingReader**. По този обект можем да се ориентираме дали възелът е от вграден XSD тип, прост потребителски или комплексен потребителски тип.
- **ValidationType** – определя типа валидация, която ще се извършва. Възможните стойности (**Auto**, **None**, **DTD**, **XDR** и **Schema**) са дефинирани в изброения тип **ValidationType**.
- **XmlResolver** – определя **XmlResolver** обекта, който се използва за извличане на външни ресурси за схемите или DTD декларациите. **XmlResolver** се прилага и при обработване на **import** и **include** елементите, съдържащи се в XSD схемите.
- **Read()** – извикването на този метод премества XML четеца в основата на **XmlValidatingReader** към следващия възел от XML дървото. В същото време валидиращият четец взима информацията за възела и го валидира спрямо избраната схема и кешираната отпреди това информация.

XmlValidatingReader не предоставя метод, който валидира съдържанието на цял XML документ. Валидиращият четец обработва възлите един по един, така както така работи XML четецът в основата му.

- **ValidationEventHandler** – всяка грешка в процеса на валидация на възлите, води до възникване на събитието **ValidationEventHandler**. Методите, които обработват това събитие трябва да имат следната сигнатура:

```
public delegate void ValidationEventHandler(
```

```
    object sender, ValidationEventArgs e);
```

Класът **ValidationEventArgs** съдържа символен низ **Message** с описание на грешката, **Exception** поле от тип **XmlSchemaException** съдържащ детайлно описание на грешката и **Severity** поле, което указва колко сериозен е проблемът.

Валидация на XML – пример

Ще разгледаме един кратък пример, който илюстрира валидацията на XML документ със средствата на класа **XmlValidatingReader**. Целта на примера е да валидираме готов XML документ по кеширана в паметта схема.

Нека се запознаем със съдържанието на XML документа, който ще валидираме:

library-valid.xml

```
<?xml version="1.0"?>
<library xmlns="http://www.nakov.com/schemas/library"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.nakov.com/schemas/library
      http://www.nakov.com/schemas/library.xsd"
   name=".NET Developer's Library">
   <book>
      <title>Programming Microsoft .NET</title>
      <author>Jeff Prosise</author>
      <isbn>0-7356-1376-1</isbn>
   </book>
   <book>
      <title>Microsoft .NET for Programmers</title>
      <author>Fergal Grimes</author>
      <isbn>1-930110-19-7</isbn>
   </book>
</library>
```

Указано е, че пространството от имена по подразбиране за този документ е **http://www.nakov.com/schemas/library** и схемата, която валидира това пространство е публикувана в Интернет на адрес **http://www.nakov.com/schemas/library.xsd**.

Ето и съдържанието на XSD схемата **library.xsd**:

library.xsd

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

```
  xmlns="http://www.nakov.com/schemas/library"
  targetNamespace="http://www.nakov.com/schemas/library">

  <xs:element name="library">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="book" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string"
          use="optional"/>
    </xs:complexType>
  </xs:element>

  <xs:element name="book">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="title"/>
        <xs:element ref="author"/>
        <xs:element ref="isbn"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="title" type="xs:string"/>
  <xs:element name="author" type="xs:string"/>
  <xs:element name="isbn" type="xs:string"/>

</xs:schema>
```

Ще резюмираме накратко XSD схемата – определят се пет глобални елемента – **library**, **book**, **title**, **author** и **isbn** – като последните три от тях са дефинирани от тип **string**, а **library** и **book** са дефинирани като комплексни типове. Структурата на **library** определя, че този елемент може да съдържа неограничен брой елементи **book** и има незадължителен атрибут **name** от тип **string**. Елементът **book** е съставен от елементи **title**, **author** и **isbn** и не дефинира атрибути. XSD схемата изрично указва, че описва елементите от пространството от имена **http://www.nakov.com/schemas/library**.

Да разгледаме процеса на валидация стъпка по стъпка:

1. Стартираме VS.NET и създаваме нов проект.
2. За да създадем валидиращ четец, необходимо е първо да инстанцираме обект от класа **XmlTextReader**, тъй като **XmlValidatingReader** не работи самостоятелно, а само върху вече създаден текстов четец:

```
XmlTextReader tr = new XmlTextReader("library-valid.xml");
```

```
XmlValidatingReader vr = new XmlValidatingReader(tr);
```

3. Дефинираме типа на предстоящата валидация и добавяме XSD схемата **library.xsd** в кеша на валидатора. Необходимо е изрично да укажем, че тя съответства на пространството от имена **http://www.nakov.com/schemas/library**. Ако не направим това, валидаторът ще търси схемата в Интернет от посочения в XML файла URL адрес **http://www.nakov.com/schemas/library.xsd**:

```
vr.Schemas.Add("http://www.nakov.com/schemas/library",
  "library.xsd");
vr.ValidationType = ValidationType.Schema;
```

4. Възможно е в процеса на валидация парсерът да открие невалиден таг, атрибут или друг проблем. Това води до възникване на събитието **ValidationEventHandler**. За да получим информация за конкретната грешка, закачаме метод-обработчик на това събитие с име **ValidationHandler(…)**. В него отпечатваме описанието и сериозността на проблема, като използваме съответните полета **Message** и **Severity** на класа **ValidationEventArgs** (параметър на метода-обработчик **ValidationHandler(…)**):

```
vr.ValidationEventHandler +=
  new ValidationEventHandler(ValidationHandler);

...

public static void ValidationHandler(object sender,
  ValidationEventArgs args)
{
  mValid = false;
  Console.WriteLine("***Validation error");
  Console.WriteLine("\tSeverity:{0}", args.Severity);
  Console.WriteLine("\tMessage:{0}", args.Message);
}
```

5. Задаваме стойност **true** на булевата член-променлива **mValid**. Тази променлива променяме единствено в метода-обработчик на събитието **ValidationEventHandler** т.е. ако XML документа е валиден, методът **ValidationHandler(…)** не се извиква нито веднъж и стойността на **mValid** остава **true**. Сега прочитаме целия XML документ възел по възел, което осигурява цялостната му валидация. Интересно е да отбележим, че не указваме изрично действие в процеса на валидация – при преминаването към всеки следващ възел, валидиращият четец си взима необходимата информация и извършва проверката:

```
    mValid = true;
    while(vr.Read())
    {
      // Do nothing, just read whole the document.
    }
```

6. След като сме прочели целия документ, остава единствено да проверим стойността на булевата променлива **`mValid`** и ако тя е непроменена, отпечатваме съобщение за успешната валидация на конзолния изход:

```
    if (mValid)
    {
      Console.WriteLine("The document is valid.");
    }
```

7. Ето как изглежда целия пример:

```
using System;
using System.Xml;
using System.Xml.Schema;

class XMLValidationDemo
{
  private static bool mValid;

  static void Main()
  {
    XmlTextReader tr = new XmlTextReader("library-valid.xml");

    XmlValidatingReader vr = new XmlValidatingReader(tr);

    vr.Schemas.Add("http://www.nakov.com/schemas/library",
      "library.xsd");
    vr.ValidationType = ValidationType.Schema;
    vr.ValidationEventHandler +=
      new ValidationEventHandler(ValidationHandler);

    mValid = true;
    while(vr.Read())
    {
      // Do nothing, just read whole the document.
    }

    if (mValid)
    {
      Console.WriteLine("The document is valid.");
    }
  }
```

```
   public static void ValidationHandler(object sender,
      ValidationEventArgs args)
   {
      mValid = false;
      Console.WriteLine("***Validation error");
      Console.WriteLine("\tSeverity:{0}", args.Severity);
      Console.WriteLine("\tMessage:{0}", args.Message);
   }
}
```

8. Ето и резултата на конзолния изход, след като компилираме и изпълним програмата:

9. Нека сега вместо документа **library-valid.xml** да валидираме документа **library-invalid.xml**. Новият XML документ изглежда по следния начин:

library-invalid.xml

```
<?xml version="1.0"?>
<library xmlns="http://www.nakov.com/schemas/library"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.nakov.com/schemas/library
      http://www.nakov.com/schemas/library.xsd"
   name=".NET Developer's Library">
   <book>
      <title name="Programming Microsoft .NET" />
      <author>Jeff Prosise</author>
      <isbn>0-7356-1376-1</isbn>
   </book>
   <book>
      <book-title>Microsoft .NET for Programmers</book-title>
      <author>Fergal Grimes</author>
      <isbn>1-930110-19-7</isbn>
   </book>
</library>
```

Единствената промяна, която трябва да направим, е при създаването на **XmlTextReader** обекта в началото на програмата:

```
   XmlTextReader tr = new XmlTextReader("library-invalid.xml");
```

10. Нека разгледаме съдържанието на конзолния изход след новата компилация и изпълнение на програмата:

Документът **library-invalid.xml** не е валиден XML документ, както можем сами да се уверим. Валидаторът открива три грешки – недекларирани атрибут **name** за елемента **title**, недекларирани елемент **book-title** и отново той не е валиден наследник на възела **book**.

Валидация на XML при DOM

Досега разгледахме процеса на валидация на XML документи с помощта на XML четци, но валидация може да се извършва и по време на конструирането на XML DOM дърво. Класът **XmlDocument** парсва цялото съдържание на подадения му XML документ в паметта чрез метода **Load(…)**. Този метод прави проверка единствено дали XML е добре дефиниран, но не го валидира спрямо схема или DTD декларация.

За да валидираме DOM дървото в процеса на изграждането му е необходимо да използваме специален конструктор на метода **Load(…)**:

```
public override void Load(XmlReader);
```

Едно XML DOM дърво може да бъде създадено по различни източници включително поток, текстов четец и име на файл. Ако заредим документа през **XMLValidatingReader** (наследник на **XmlReader**), постигаме валидация на DOM дървото едновременно с неговото изграждане. Ще скицираме необходимия за целта сорс код:

```
XmlDocument doc = new XmlDocument();
XmlTextReader tr = new XmlTextReader("Sample.xml");
XmlValidatingReader valReader = new XmlValidatingReader(tr);
valReader.ValidationType = ... ;
valReader.ValidationEventHandler += ... ;
doc.Load(valReader);
```

XPath

До момента разгледахме доста неща, свързани с работата с XML, но това не е всичко. XML технологиите са много и ролята им в съвременното програмиране постоянно нараства. Затова ще разгледаме още няколко от тези технологии – XPath и XSLT.

Ще започнем от технологията XPath, която има широко приложение при извличане на информация от XML документи и се използва като съставна част в други XML технологии.

Описание на езика

XPath спецификацията се появява скоро след публикуването на XML 1.0 стандарта и представлява W3C утвърден език за адресиране на части от XML документи. XPath изразите приличат на описания на пътища от файловата система, съставени от имена на файлове и директории (оттам идва и името на езика XPath).

Езикът XPath обслужва XML документи, но синтаксисът му не е XML-базиран.

Ще дадем много кратко описание на езика XPath без да навлизаме в подробности, след което ще дадем няколко примера.

Какво представлява един XPath израз?

XPath изразите съдържат описания на пътища до възли и критерии, на които тези възли трябва да отговарят. Описанията могат да бъдат относителни или абсолютни и това определя в какъв контекст се оценява една XPath заявка.

Един XPath израз винаги се оценява в определен контекст (контекстен възел, контекстно множество от възли).

В началото контекстният възел се определя от приложението и представлява начална точка за XPath заявката. На всяка стъпка от описания XPath път контекстният възел приема стойността на текущия възел. Възлите, които имат отношение към частта на XPath заявката, изпълнявана в момента, образуват контекстно множество. Множеството възли, което се връща като краен резултат, включва само част от тези възли, които отговарят на зададени допълнителни условия.

XPath стъпка

Един XPath път се състои от една или повече **локационни стъпки** (разделени със символ "/"). Всяка стъпка се състои от ос (незадължителен елемент), тест на възли и предикат (също незадължителен елемент):

```
ос::тест-на-възли[предикат]
```

Ос на XPath стъпката

Оста определя йерархичната връзка между контекстния възел и възлите, избирани на една локационна стъпка (т. е. определя контекстното множество възли за всяка стъпка). Ако XPath стъпка няма зададена ос, по подразбиране в контекстното множество възли участват преките наследници на контекстния възел. Нека да разгледаме някои възможни стойности за ос на XPath стъпка:

- **self(.)** – включва самия контекстен възел
- **child** – включва преките наследници на контекстния възел
- **parent(..)** – включва родителят на контекстния възел
- **descendant** – включва възлите от поддървото с корен контекстния възел (участват само възлите от тип елемент, текст и инструкция за обработка, но не и коментари или атрибути)
- **descendant-or-self(//)** – разновидност на **descendant**, която включва към дървото и самия контекстен възел
- **ancestor** – включва предшествениците на контекстния възел в йерархията чак до коренния елемент
- **ancestor-or-self** – разновидност на **ancestor**, включва към множество и самия контекстен възел
- **attribute(@)** – включва атрибутите на контекстния възел, ако той е от тип елемент

При използване на съкратената форма за осите (указана в скоби по-горе) не се използва разделител "::" между оста и теста на възли.

Тест на възли

Тестът на възли е основан на възли израз, който се оценява за всеки възел в контекстното множество. Ако тестът върне положителен резултат, възелът остава в множеството, а в противен случай се премахва оттам. Обикновено тестът на възли се състои от описание на път до даден възел и връща положителен резултат, ако пътят съществува в текущия контекст. Той може да съдържа XPath функции:

- **text()** – връща текстовото съдържание на контекстния възел
- **comment()** – връща всички наследници на контекстния възел от тип коментар

- **processing instruction()** – връща всички наследници на контекстния възел, които са от тип инструкция за обработка
- **node()** – връща всички наследници на контекстния възел

Предикат

Предикатът е незадължителен логически израз, който се прилага като допълнителен филтър върху текущото множество от възли, получено след изпълнението на възловия тест. Той също може да съдържа XPath функции (например **count(…)** връща броя на възлите в множеството подадено като параметър). Една XPath стъпка може да има повече от един предикат, като те се записват един след друг подобно на индекси на многомерен масив.

XPath изразите в действие

Нека разгледаме няколко практически XPath примери, които ще ни помогнат да разберем начина на работа на езика XPath:

- **/** – адресира коренния елемент на документа
- **/someNode** – адресира всички възли с име **someNode**, преки наследници на корена
- **/books/book** – адресира всички възли **book**, наследници на възел **books** (**books** от своя страна е пряк наследник на корена)
- **books/book** – адресира всички възли **book**, наследници на възел **books** (няма ограничения за местоположението в документа на елемента **books**)
- **/books/book[price<"10"]/author** – адресира всички автори (**/books/book/author**), чиито книги имат цена по-малка от 10
- **/items/item[@type="food"]** – адресира всички възли с име **item**, които имат атрибут **type** със стойност **"food"** и са наследници на възел **items**, пряк наследник на корена на документа

Сега ще демонстрираме един по сложен пример за финал, като ще опишем процеса на адресация на възлите стъпка по стъпка – **/book/chapter[3]/para[last()][@lines > 10]**:

1. Адресираме елементите **book**, преки наследници на корена на документа.
2. Измежду възлите-наследници на **book** с име **chapter** адресираме само третите по ред.
3. Измежду техните възли-наследници **para** адресираме само последните по ред с това име, които имат атрибут **lines** със стойност поголяма от 10.

XPath, XSLT и XPointer

XPath често пъти се интегрира с XSLT и XPointer. Езикът XPath обикновено се използва за претърсване на XML DOM източник на данни и служи за филтриране на възлите, върху които се прилага определена XSL трансформация. XPath се налага все повече и при XPointer – формализъм за идентифициране на фрагменти от XML документи. Ето един пример за интеграция на XPath и XPointer:

```
library.xml#xpointer(/library/book[isbn='1-930110-19-7'])
```

Този израз сочи към възел **book**, наследник на **library**, който има наследник **isbn** със стойност **'1-930110-19-7'**.

XPath в .NET Framework

.NET Framework осигурява пълна поддръжка за езика XPath чрез класовете в пространството от имена **System.Xml.XPath**. Имплементацията на XPath е основана на езиков парсер и оценяващ модул. Общата архитектура прилича на тази при заявките към бази данни – както SQL командите, XPath изразите се подготвят предварително и се подават на оценяващ модул по време на изпълнение на програмата.

XPath и XmlNode

Средствата за работа с XPath в .NET Framework, освен през класовете от пространството **System.Xml.XPath** (програмен интерфейс, основан на концепцията за XPath навигатор), са достъпни директно през XML DOM модела (класът **XmlNode**).

Извикването на XPath изрази през класа **XmlNode** винаги се извършва в контекст на вече съществуваща инстанция на класа **XmlDocument**. Програмният интерфейс на този подход носи особеностите на стария COM-базиран MSXML стил на програмиране, популярен доскоро при Win32 приложенията за работа с XML. XML DOM поддръжката за XPath изрази улеснява преминаването от MSXML към .NET Framework стила и предоставя вграден механизъм за търсене на възли в зареден в паметта XML документ.

XPath и XmlNode – методи

XPath може да се използва директно от класа **XmlNode** и всички негови наследници през следните методи:

- **SelectNodes(string xPathQuery)** – връща списък от всички възли, които съответстват на зададения XPath израз.
- **SelectSingleNode(string xPathQuery)** – връща първия възел, който съответства на зададения XPath израз.

XPath и XmlNode – пример

Следният кратък пример илюстрира работата с XPath и **XmlNode** за търсене на възли в един XML документ.

Ще използваме следния работен документ **items.xml**:

items.xml

```
<?xml version="1.0" encoding="windows-1251"?>
<items>
   <item type="beer">
      <name>Загорка</name>
      <price>0.54</price>
   </item>
   <item type="food">
      <name>кебапчета</name>
      <price>0.48</price>
   </item>
   <item type="beer">
      <name>Каменица</name>
      <price>0.56</price>
   </item>
</items>
```

Целта на демонстрацията е да открием имената на всички стоки от тип **"beer"** в този документ.

Нека разгледаме необходимите стъпки, за да постигнем това:

1. Стартираме VS.NET и създаваме нов проект.
2. Както споменахме вече, съвместната работа на XPath и **XmlNode** изисква съществуваща инстанция на **XmlDocument**. Затова първо създаваме нов **XmlDocument** и зареждане в него XML документа **items.xml**:

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load("items.xml");
```

3. Сега вече можем да подготвим самия XPath израз, който описва всички имена на стоки от тип бира в зададения XML документ - търсим всички възли **name**, наследници на тези възли **item**, чиито атрибут **type** има стойност **"beer"**. Възлите **item** от своя страна трябва да са преки наследници на възли **items**, закачени за корена на документа:

```
string xPathQuery = "/items/item[@type='beer']/name";
```

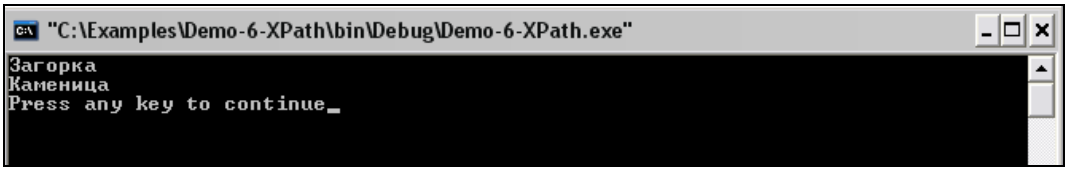
4. Извличаме всички възли, които отговарят на XPath израза и ги записваме в **XmlNodeList**:

```
XmlNodeList beerNamesList = xmlDoc.SelectNodes(xPathQuery);
```

5. Обхождаме елементите на списъка и отпечатваме имената на стоките, които извлякохме от XML документа:

```
foreach (XmlNode beerName in beerNamesList)
{
   Console.WriteLine(beerName.InnerText);
}
```

6. Ето съдържанието на конзолния изход след компилация и изпълнение на програмата:



Пространството System.Xml.XPath

Същинският програмен интерфейс, осигуряващ функционалност за обработка на XPath изрази, е реализиран от класа **`XPathNavigator`**. В действителност извикванията към методите **`SelectSingleNode(…)`** и **`SelectNodes(…)`** от класа **`XmlNode`** вътрешно също създават обект навигатор в процеса на своята работа.

Класът XPathNavigator

`XPathNavigator` е абстрактен клас, който дава възможност за навигация между отделните възли в един XML документ и изпълнение на XPath заявки върху тях.

Важно е да отбележим, че навигаторът е съвсем отделен компонент от документния клас. **`XPathNavigator`** работи единствено върху специална категория документни класове, известни като **XPath хранилища за данни (XPath data store)**. Тези класове представят съдържанието си под формата на XML и позволяват изпълнение на XPath заявки върху тях. Един клас в .NET Framework придобива такава възможност, имплементирайки интерфейса **`IXPathNavigable`**. Този интерфейс съдържа единствен метод **`CreateNavigator()`**, който създава инстанция на специализиран за конкретния документ навигатор (наследник на абстрактния клас **`XPathNavigator`**).

В .NET Framework вградените XPath хранилища са само три – **`XmlDocument`**, **`XPathDocument`** и **`XmlDataDocument`**.

XPathNavigator – по-важни методи

- **`Select(…)`** – връща множество от възли, отговарящо на зададен XPath израз. Контекстът, в който се оценява XPath заявката, е позицията на навигатора при извикването на метода.
- **`SelectAncestors(…)`** – връща всички предшественици на текущия възел. Резултатното множество може да се ограничи с допълнителни филтри за име на възел и пространство от имена.
- **`SelectChildren(…)`** – връща всички преки наследници на текущия възел. Резултатното множество може да се ограничи с допълнителни филтри за име на възел и пространство от имена. Атрибутите и пространствата от имена не участват в резултата.
- **`SelectDescendants(…)`** - връща всички наследници на текущия възел. Резултатното множество може да се ограничи с допълнителни филтри за име на възел и пространство от имена. Атрибутите и пространствата от имена не участват в резултата.
- **`MoveTo(…)`** – придвижва навигатора до позицията, определена от зададения като параметър **`XPathNavigator`** обект.
- **`MoveToNext(…)`** – придвижва се до следващия наследник на текущия възел.
- **`MoveToParent(…)`** – придвижва се до родителя на текущия възел.
- **`Compile(…)`** – компилира XPath израз.
- **`Matches(…)`** – определя дали текущия възел отговаря на зададен XPath израз

XPathNodeIterator

Всеки път, когато един XPath израз поражда резултатно множество от възли (при извикване на **`Select`** методите), навигаторът връща като резултат нов обект от тип итератор на възли. Итераторът предоставя интерфейс за навигация в масив от възли. Базовият клас за всеки XPath итератор е класът **`XPathNodeIterator`**.

Функциите на един итератор лесно могат да бъдат припокрити от един XPath навигатор, но .NET Framework съзнателно предоставя функционалността им в отделни компоненти. Избраният подход на разделяне на програмните интерфейси на навигаторите и итераторите е реализиран с цел осигуряване на по-лесен достъп и обработка на XPath заявки от различни среди – XML DOM, XPath и XSLT.

XPathNodeIterator – същност

Класът **XPathNodeIterator** няма публичен конструктор и може да бъде създаван единствено от обект навигатор. Итераторът осигурява еднопосочен достъп до възлите от една XPath заявка. Итераторът не кешира информация за възлите, които обхожда – той е просто индексатор, работещ върху обект от тип навигатор, който управлява XPath заявката.

XPathNodeIterator – методи и свойства

- **MoveNext()** – придвижва се до следващия възел в множеството избрани възли на навигатора.
- **Clone()** – връща дълбоко копие на текущия **XPathNodeIterator**
- **Count** – връща броя на елементите от първо ниво (не отчита наследниците) в множеството от възли.
- **Current** – връща референция към навигатор с корен във възела, разположен на текущата позиция на итератора.
- **CurrentPosition** – връща индекса на текущия възел, избран от итератора.

XPathNavigator – пример

Ще разгледаме кратък пример, който илюстрира работата с **XPathNavigator** за обхождане и промяна на части от един XML документ. Целта на демонстрацията е да намалим цената на всички стоки от тип "бира" в XML документа с 20%.

Работният документ, който ще използваме изглежда по следния начин:

items.xml

```
<?xml version="1.0" encoding="windows-1251"?>
<items>
  <item type="beer">
    <name>Загорка</name>
    <price>0.54</price>
  </item>
  <item type="food">
    <name>кебапчета</name>
    <price>0.48</price>
  </item>
  <item type="beer">
    <name>Каменица</name>
    <price>0.56</price>
  </item>
</items>
```

Нека разгледаме необходимите действия стъпка по стъпка:

1. Стартираме VS.NET и създаваме нов проект.
2. За да получим инстанция на **XPathNavigator** обект е необходимо първо да инстанцираме нов обект от тип **XmlDocument** с работния документ **items.xml**:

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load("items.xml");
```

3. Създаваме инстанция на обект от тип **XPathNavigator**:

```
XPathNavigator nav = xmlDoc.CreateNavigator();
```

4. За да намалим цената на бирата с 20%, първо трябва да имаме лесен начин за навигация до цените на стоките от тип бира. XPath изразът, който ще използваме, търси всички възли **price**, наследници на тези възли **item**, чиито атрибут **type** има стойност **"beer"**. Възлите **item** от своя страна трябва да са преки наследници на възли **items**, закачени за коренния елемент на документа. Изпълняваме тази XPath заявка върху **XPathNavigator** обекта и получаваме **XPathNodeIterator**:

```
string xPathQuery = "/items/item[@type='beer']/price";
XPathNodeIterator iter = nav.Select(xPathQuery);
```

5. С помощта на итератора обхождаме възлите, върнати като резултат от XPath заявката върху навигатора. Поради стремежа за разделяне на функционалността на итераторите и навигаторите, процесът на взимане стойността за всеки възел изглежда малко неестествен в началото. От итератора получаваме **XPathNavigator** обект с корен текущия възел. За да получим **XmlNode** инстанция от навигатора, извикваме метода **GetNode()** от интерфейса **IHasXmlNode** (имплементиран от **XpathNavigator**):

```
while (iter.MoveNext())
{
   XPathNavigator currentNode = iter.Current;
   XmlNode xmlNode = ((IHasXmlNode) currentNode).GetNode();
   ...
}
```

6. Разглежданият пример чете и променя стойности на десетични числа. В XML документа те са форматирани с десетична точка, но винаги има вероятност текущата активна култура на компютъра, където изпълняваме програмата, да е различна и да форматира числата с десетична запетая (например българската култура). За да се подсигурим, че парсването на числата ще протече безпроблемно и няма да предизвика изключение от тип **FormatException**, добре е

да използваме инвариантната (културно-необвързана) култура, достъпна чрез свойството **`CultureInfo.InvariantCulture`**.

7. След като вече имаме достъп до цената под формата на **`XmlNode`**, взимаме стойността като низ и я парсваме като десетично число, с помощта на културата **`CultureInfo.InvariantCulture`**. Тази стойност намаляваме с 20% и я записваме като нова стойност на текущия възел:

```
public const decimal DISCOUNT = (decimal) 0.20;
...

while (iter.MoveNext())
{
   ...
   string priceStr = xmlNode.InnerText;
   decimal price = Decimal.Parse(priceStr,
      CultureInfo.InvariantCulture);
   price = price * (1 - DISCOUNT);
   xmlNode.InnerText = price.ToString(
      CultureInfo.InvariantCulture);
}
```

8. Единственото, което остава, е да запазим новия XML документ:

```
xmlDoc.Save("itemsNew.xml");
```

9. Ето как изглежда пълният сорс код на примера:

```
using System;
using System.Xml;
using System.Xml.XPath;
using System.Globalization;

class XPathNavigatorDemo
{
  public const decimal DISCOUNT = (decimal) 0.20;

  static void Main()
  {
    XmlDocument xmlDoc = new XmlDocument();
    xmlDoc.Load("../../../xml-files/items.xml");

    CultureInfo numberFormat = new CultureInfo("en-US");

    XPathNavigator nav = xmlDoc.CreateNavigator();
    string xPathQuery = "/items/item[@type='beer']/price";
    XPathNodeIterator iter = nav.Select(xPathQuery);

    while (iter.MoveNext())
```

```
      {
         XPathNavigator currentNode = iter.Current;
         XmlNode xmlNode = ((IHasXmlNode) currentNode).GetNode();
         string priceStr = xmlNode.InnerText;
         decimal price = Decimal.Parse(priceStr, numberFormat);
         price = price * (1 - DISCOUNT);
         xmlNode.InnerText = price.ToString(numberFormat);
      }

      xmlDoc.Save("itemsNew.xml");
   }
}
```

10. Можем да се уверим, че променена е само цената на бирата, като разгледаме документа **itemsNew.xml** след компилация и изпълнение на програмата:

itemsNew.xml

```
<?xml version="1.0" encoding="windows-1251"?>
<items>
  <item type="beer">
    <name>Загорка</name>
    <price>0.432</price>
  </item>
  <item type="food">
    <name>кебапчета</name>
    <price>0.48</price>
  </item>
  <item type="beer">
    <name>Каменица</name>
    <price>0.448</price>
  </item>
</items>
```

Класът XPathDocument

Класът **XPathDocument** осигурява силно оптимизиран кеш на XML документи в паметта за еднопосочна работа с XPath и XSLT в режим само за четене. Този клас е проектиран специално с цел да служи за контейнер на XPath данни и не осигурява никаква информация за възлите, които съдържа. Класът **XPathDocument** създава поддържаща мрежа от референции към възли, която позволява на XPath навигатора да работи бързо и ефективно. **XPathDocument** не зачита XML DOM спецификациите и съдържа единствен метод **CreateNavigator()** (от интерфейса **IXPathNavigable**). Удобен е при обработката на големи XML документи.

Работа с XPathDocument – пример

Търсенето на имената на всички стоки от тип бира, което демонстрирахме по-горе със средствата на XPath и **XmlNode**, лесно може да се осъществи с помощта на оптимизирания клас **XPathDocument**. Тъй като работата на класа **XPathNavigator** е напълно независима от документния клас, стъпките отново са същите, като в току-що разгледаната демонстрация:

1. Зарежда се документен клас (XPath хранилище за данни) – в този случай **XPathDocument**.
2. С негова помощ се създава навигатор.
3. Изпълнява се XPath заявка върху този навигатор (в този случай компилирана).
4. С помощта на итератор, получен от изпълнението на заявката, се обхожда множеството от възли и се изписва стойността им.

```
XPathDocument doc = new XPathDocument("items.xml");

XPathNavigator nav = doc.CreateNavigator();

XPathExpression expr = nav.Compile(
   "/items/item[@type='beer']/name");
XPathNodeIterator iter = nav.Select(expr);

while (iter.MoveNext())
{
   XPathNavigator currentNode = iter.Current;
   Console.WriteLine(currentNode.Value);
}
```

Класът XPathExpression

Класът **XPathExpression** представлява компилиран XPath израз – енкапсулация на XPath описание на път и контекст, в който ще се оценява то. **XPathExpression** няма публичен конструктор и не може да бъде създаван директно.

Компилиране на XPath изрази

XML DOM методите **SelectSingleNode(…)** и **SelectNodes(…)**, както и **Select** методите на класа **XPathNavigator** позволяват задаването на XPath заявката под формата на чист текст. XPath изразите, обаче, се изпълняват само в компилирана форма, затова тези методи прозрачно извършват компилация на XPath изразите преди да ги обработят. Потребителите могат сами да създават компилиран XPath израз с метода **XpathNavigator.Compile(…)**. Използването на компилиран XPath израз има някои предимства:

- Преизползваемост (при многократна работа с един и същи XPath израз компилация се извършва само веднъж).
- Компилацията на XPath израз позволява предварително да се знае типа на върнатата стойност (изброеният тип **`XPathResultType`**).

Компилираните XPath изрази могат да се използват като параметри за някои от методите на класа **`XPathNavigator`**, между които **`Select(…)`**, **`Evaluate(…)`**, **`Matches(…)`**.

XML DOM методите SelectSingleNode(…) и SelectNodes(…) не могат да приемат компилиран XPath израз като параметър. (Вътрешно те създават инстанция на XPathNavigator, която отново компилира текстовия XPath израз, но в този случай нямаме преизползваемост).

XSLT

След като разгледахме XPath технологията, нека се запознаем и с XSL трансформациите на XML документи.

Технологията XSLT

XSLT (Extensible Stylesheet Language Transformations) е език, който позволява трансформиране на XML документи в XML или друг текстов формат в зависимост от зададени правила.

XSLT е подмножество на езика XSL (Extensible Stylesheet Language – език за описание представянето на XML-форматирани документи), като в началото се използва за трансформации на XML елементи в комплексни стилове (например вложени таблици и индекси).

Какво представлява XSLT?

Една **XSL трансформация** е процес, при който даден XML документ се преобразува в друг текстов документ. За целта се използва XSLT шаблон, по който се извършва трансформацията.

XSL шаблон за трансформация – същност

Един **XSLT шаблон** представлява на практика поредица от шаблонни елементи. Всеки шаблон приема като вход един или повече елементи от входния XML документ и връща текстов изход въз основа на свои литерали и трансформация на приетите входни параметри.

XSLT процесорът обработва документите последователно, но разчита на XPath извиквания за извличането на възли с определени характеристики. Резултатът от една трансформация може да бъде XML документ (в частност и XHTML), HTML страница или документ във всеки един текстово-базиран формат, който отговаря на правилата, описани от трансформацията.

XSL шаблон за трансформация – пример

Нека разгледаме как в действителност изглежда един XSL шаблон за трансформация:

library-xml2html.xsl

```
<?xml version="1.0" encoding="windows-1251"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<body>
  <h1>Моята библиотека</h1>
  <table bgcolor="#E0E0E0" cellspacing="1">
    <tr bgcolor="#EEEEEE">
      <td><b>Заглавие</b></td>
      <td><b>Автор</b></td>
    </tr>
    <xsl:for-each select="/library/book">
      <tr bgcolor="white">
        <td><xsl:value-of select="title"/></td>
        <td><xsl:value-of select="author"/></td>
      </tr>
    </xsl:for-each>
  </table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Засега няма да навлизаме в техническите подробности и само ще споменем, че можем да използваме този шаблон за трансформация на информацията за малка домашна библиотека от XML в HTML формат.

По-важни XSLT конструкции

Конструкциите на езика XSLT се състоят от специални тагове. Те представят отделните операции, които могат да се извършват върху markup от входния документ или върху подадените на трансформацията параметри:

- **<xsl:template match="XPath-израз">…</xsl:template>** – замества, зададената с XPath израз, част от документа с тялото на конструкцията.
- **<xsl:value-of select="XPath-израз" />** – извлича стойността на зададения XPath израз (само първото намерено съответствие).
- **<xsl:for-each select="XPath-израз">…</xsl:for-each>** – замества всеки възел, отговарящ на дадения XPath израз с тялото на конструкцията.

- **<xsl:if test="XPath-израз">…</xsl:if>** – прилага тялото на конструкцията само, ако XPath изразът се оцени с положителна булева стойност.
- **<xsl:sort select="XPath-израз" />** – в **xsl:for-each** конструкции сортира по стойността на даден XPath израз.

XSLT и .NET Framework

.NET Framework осигурява пълна XSLT поддръжка чрез класовете от пространството **System.Xml.Xsl**. Основен клас при работата с XSLT е **XslTransform**, който представлява имплементация на XSLT процесор за .NET Framework. Работата с този клас протича винаги в две стъпки – XSLT шаблонът първо се зарежда в процесора и едва тогава се извършват трансформации с него.

Класът **XslTransform** поддържа само версия 1.0 на XSLT спецификацията. Един шаблон декларира съвместимост с тази версия, като включва следното пространство от имена:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

Важно е да се отбележи, че атрибутът version е задължителен, за да се осигури коректността на XSLT документа.

XslTransform – методи

Методите от класа **XslTransform**, които най-често се използват, са следните:

- **Load(…)** – зарежда XSL шаблон за трансформацията
- **Transform(…)** – извършва трансформация на даден XML документ. Приема като вход име на XML файл, **XPathNavigator** или **IXPathNavigable**. Записва изхода в XML файл, поток или **XmlWriter**.

XSL трансформация – пример

Следният кратък пример илюстрира работата с **XslTransform** класа за преобразуване на XML документ по даден XSLT шаблон. Целта, която си поставяме е да трансформираме съдържанието на XML документа **library.xml** в HTML формат.

Ето как изглежда документа **library.xml**:

library.xml

```
<?xml version="1.0"?>
<library name=".NET Developer's Library">
  <book>
```

```
    <title>Programming Microsoft .NET</title>
    <author>Jeff Prosise</author>
    <isbn>0-7356-1376-1</isbn>
  </book>
  <book>
    <title>Microsoft .NET for Programmers</title>
    <author>Fergal Grimes</author>
    <isbn>1-930110-19-7</isbn>
  </book>
</library>
```

Ето и съдържанието на XSL шаблона **library-xml2html.xsl**, който описва правилата за трансформацията:

library-xml2html.xsl

```
<?xml version="1.0" encoding="windows-1251"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<body>
  <h1>Моята библиотека</h1>
  <table bgcolor="#E0E0E0" cellspacing="1">
    <tr bgcolor="#EEEEEE">
      <td><b>Заглавие</b></td>
      <td><b>Автор</b></td>
    </tr>
    <xsl:for-each select="/library/book">
      <tr bgcolor="white">
        <td><xsl:value-of select="title"/></td>
        <td><xsl:value-of select="author"/></td>
      </tr>
    </xsl:for-each>
  </table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Шаблонът определя, че коренният елемент на входния документ (на практика това адресира целия документ, защото всички други възли са наследници на коренния елемент) трябва да се замести с тялото на HTML конструкцията, дефинирана от шаблона.

Като оставим на страна стандартните HTML елементи, интерес за нас представляват XSLT таговете **xsl:for-each** и **xsl:value-of**. Тагът **<xsl:for-each select="/library/ book">** замества всеки възел от входния документ, който отговаря на зададения XPath израз с ред от таблица,

чиито колони се инициализират със стойности, извлечени от наследниците на възлите **/library/book** (**xsl:value-of** конструкциите).

Нека сега разгледаме необходимите стъпки за реализиране на програма, която извършва XSL трансформацията:

1. Създаваме нов обект от тип **XslTransform**:

```
XslTransform xslt = new XslTransform();
```

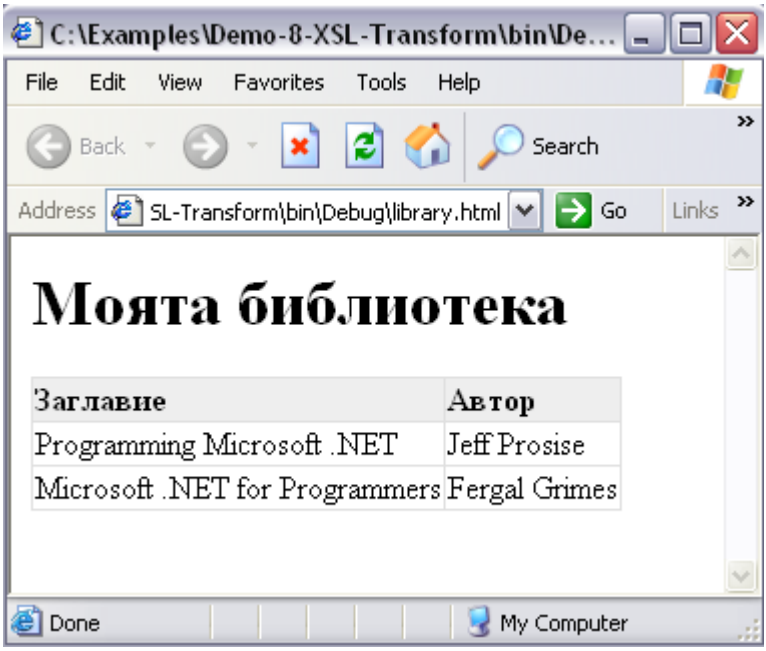
2. Зареждаме XSLT шаблона, описващ трансформацията:

```
xslt.Load("library-xml2html.xsl");
```

3. Извършваме трансформацията, като запазваме резултата във файла **library.html**. Тъй като не използваме външни XML ресурси, инициализираме третия параметър **XmlResolver** с **null**:

```
xslt.Transform("library.xml", "library.html", null);
```

4. Ето как изглежда резултатният файл **library.html** след компилация и изпълнение на програмата:



Трансформация на XML в чист текст – пример

Както знаем, XSL трансформациите могат да преобразуват даден XML документ не само в друг XML документ, но и в произволен текстов формат. За да илюстрираме това, да си поставим следната задача: Даден е XML документът:

example.xml

```
<?xml version="1.0" encoding="utf-8"?>
<values>
    <value>1</value>
    <value>2</value>
    <value>3</value>
</values>
```

Да се напише XSL шаблон, който трансформира този документ в следния текстов вид:

example.txt

```
1<2<3
```

По подразбиране XSL трансформациите преобразуват XML документ в друг XML документ. За да преобразуваме XML документ в текст, трябва да укажем в XSL шаблона следната опция:

```
<xsl:output method="text" />
```

Тя указва на XSL трансформатора да генерира изхода като чист текст вместо като XML. Сега вече за да решим поставената задача, можем да използваме следния XSL шаблон:

xml2text.xsl

```
<?xml version="1.0" encoding="utf-8" ?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text" />
<xsl:template match="/">
  <xsl:for-each select="//values">
    <xsl:for-each select="./*">
      <xsl:if test="position()>1">
        <xsl:text>&lt;</xsl:text>
      </xsl:if>
      <xsl:value-of select="." />
    </xsl:for-each>
  </xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

С това завършваме прегледа на средствата за работа с XML в .NET Framework. Препоръчваме Ви да не пропускайте секцията с практически упражнения, която ще Ви помогне да проверите и затвърдите познанията си относно имплементацията на XML технологиите в .NET.

Упражнения

1. Какво представлява езикът XML? За какво служи? Кога се използва?
2. Създайте XML документ **students.xml**, който съдържа структурирано описание на студенти. За всеки студент трябва да има информация за имената му, пол, рождена дата, адрес, телефон, email, курс, специалност, факултетен номер, ВУЗ, факултет, положени изпити (име на изпит, преподавател, оценка), невзети изпити, среден успех, дата на приемане във ВУЗ и очаквана дата на завършване (година и месец).
3. Какво представляват пространствата от имена в XML документите? За какво служат? Кога се използват?
4. Променете файла **students.xml** и му добавете пространство от имена по подразбиране **"urn:students"**.
5. Какво представляват XML схемите? По какво си приличат и по какво се различават DTD, XSD и XDR схемите?
6. С помощта на VS.NET създайте подходяща XSD схема за валидация на документа **students.xml**. Редактирайте генерираната схема, като внимателно съобразите всяко поле от описанието на един студент от какъв тип трябва да бъде, дали трябва да е задължително или по избор и какви са ограниченията над валидните му стойности.
7. Чрез редактора на VS.NET дефинирайте XSD схема за описание на музикален каталог. Каталогът трябва да съдържа съвкупност от албуми на различни изпълнители. За всеки албум трябва да са дефинирани: наименование, автор, година на издаване, продуцентска къща, цена и списък на песните. Всяка песен трябва да се описва със заглавие и продължителност.
8. Създайте примерен XML файл **catalog.xml**, отговарящ на описаната XSD схема. Свържете файла **catalog.xml** със съответната му схема и го валидирайте по нея с помощта на VS.NET.
9. Напишете програма, която с помощта на DOM парсера и класовете **XmlDocument** и **XmlNode** извлича от **students.xml** имената на всички студенти, които имат поне 2 невзети изпита.
10. Напишете програма, която с помощта на DOM парсера и чрез използване на хеш-таблица намира и извлича всички различни автори на музика, които се срещат във файла **catalog.xml**. За всеки автор трябва да се отпечата броя на албумите му в каталога.
11. Напишете програма, която с помощта на DOM парсера добавя даден изпит в списъка с невзетите изпити за всеки студент от файла **students.xml**. Изпитът е с фиксирано заглавие **"Нов изпит"** с преподавател **"Нов преподавател"** и трябва да се добавя само ако не се среща в списъка от взетите и в списъка от невзетите изпити за студента.

12. Напишете програма, която с помощта на DOM парсера изтрива от файла **catalog.xml** всички албуми, които струват повече от 20 лв.

13. В текстов файл в някакъв предварително известен формат са записани трите имена, адреса и телефона на даден човек. Напишете програма, която с помощта на DOM парсера създава нов XML документ, който съдържа тези данни в структуриран вид.

14. Напишете програма, която с помощта на парсера **XmlReader** извлича всички заглавия на албуми от файла **catalog.xml**.

15. Напишете програма, която с помощта на парсера **XmlReader** извлича и отпечатва за всеки студент от файла **students.xml** списък от имената на всички преподаватели, при които студентът е взел успешно някакъв изпит.

16. В текстов файл в някакъв предварително известен формат са записани трите имена, адресът и телефонът на даден човек. Напишете програма, която с помощта на класа **XmlWriter** създава нов XML документ, който съдържа тези данни в структуриран вид.

17. Напишете програма, която с помощта на класовете **XmlReader** и **XmlWriter** прочита файла **catalog.xml** и създава файла **album.xml**, в който записва по подходящ начин имената на всички албуми и техните автори.

18. Напишете програма, която претърсва зададена директория от твърдия диск и записва в XML файл нейното съдържание заедно с всичките ѝ поддиректориите. Използвайте таговете **<file>** и **<dir>** с подходящи атрибути. За генерирането на XML документа използвайте класа **XmlWriter**.

19. Напишете програма, която валидира файла **students.xml** по съответната му XSD схема.

20. Напишете програма, която с помощта на DOM модела и подходящи XPath заявки за всеки студент от документа **students.xml** извлича всичките му оценки и средния му успех и проверява дали успехът е правилно изчислен.

21. Напишете програма, която с помощта на класа **XPathNavigator** и подходящи XPath заявки извлича от файла **catalog.xml** цените на всички албуми, издадени преди 5 или повече години.

22. Напишете програма, която с помощта на XPath заявки върху DOM дървото на документа **students.xml** намира за всеки студент всички изпити, които той е взел с оценка среден (3) и променя оценката му на отличен (6).

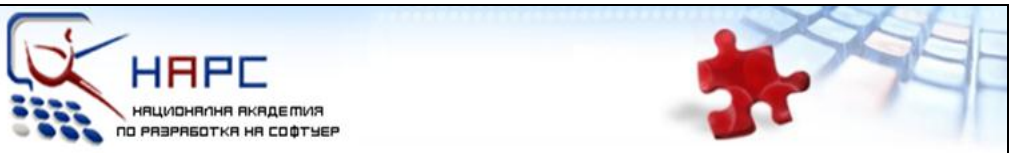
23. Създайте подходящ XSL шаблон, който преобразува файла **catalog.xml** в XHTML документ, подходящ за разглеждане от

стандартен уеб браузър. Напишете програма, която прилага шаблона с помощта на класа **`XslTransform`**.

24. Създайте XSL шаблон, който приема като вход документа **`students.xml`** и генерира като резултат друг XML документ, съдържащ само имената и факултетните номера на всички студенти. Напишете програма, която прилага шаблона с помощта на класа **`XslTransform`**.

Използвана литература

1. Светлин Наков, Работа с XML – http://www.nakov.com/dotnet/lectures/Lecture-12-Working-with-XML-v1.0.ppt
2. Стоян Йорданов, Работа с XML – http://www.nakov.com/dotnet/2003/lectures/Working-with-XML.doc
3. MSDN Training, Introduction to XML and the Microsoft® .NET Platform (MOC 2500A)
4. XML in 10 points – http://www.w3.org/XML/1999/XML-in-10-points
5. MSDN Library, XML Fundamentals: Understanding XML – http://msdn.microsoft.com/library/en-us/dnxml/html/UnderstXML.asp
6. XML Fundamentals: Understanding XML Namespaces – http://msdn.microsoft.com/XML/Understanding/Fundamentals/default.aspx?pull=/library/en-us/dnxml/html/xml_namespaces.asp
7. XML Fundamentals: Understanding XML Schema – http://msdn.microsoft.com/XML/Understanding/Fundamentals/default.aspx?pull=/library/en-us/dnxml/html/understandxsd.asp
8. William J. Pardi, "XML in Action", 1999, Microsoft Press, ISBN 0735605629
9. Erik T. Ray, "Learning XML, 2nd Edition", 2003, O'Reilly, ISBN 0596004206
10. Dino Esposito, "Applied XML Programming for Microsoft .NET", 2003, Microsoft Press, ISBN 0735618011
11. Niel M. Bornstein, ".NET and XML", 2003, O'Reilly, ISBN 0596003978



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 13. Релационни бази от данни и MS SQL Server

Необходими знания

- Базови познания за .NET Framework
- Базови познания за езика C#
- Базови познания по XML технологии
- Познания по релационни бази от данни
- Познания по езика SQL

Съдържание

- **Релационни бази от данни**
  - Модели на базите от данни
  - Релационни бази от данни. RDBMS системи
  - Таблици, връзки, множественост на връзките, E/R диаграми
  - Релационна схема. Нормализация
  - Ограничения (constraints)
  - Индекси
  - Езикът SQL
  - Изгледи (views)
  - Съхранени процедури в базата (stored procedures)
  - Тригери (triggers)
  - Транзакции и изолация
- **Въведение в MS SQL Server**
  - Компоненти на SQL Server 2000
  - Програмни среди и инструменти за разработка
  - Въведение в T-SQL
  - Data Definition Language (DDL) команди
  - Data Manipulation Language (DML) команди
  - Съединения между таблици и агрегиращи функции
  - Database Console Commands (DBCC) команди

- Съхранени процедури
- Транзакции в SQL Server
- Пренасяне на база данни

В тази тема ...

В настоящата тема ще разгледаме системите за управление на релационни бази от данни. Ще обясним свързаните с тях понятия като таблици, връзки, релационна схема, нормализация, изгледи, ограничения, транзакции, съхранени процедури и тригери. Ще се запознаем накратко с езика SQL, използван за манипулиране на релационни бази от данни.

След въведението в проблематиката на релационните бази от данни ще направим кратък преглед на Microsoft SQL Server, като типичен представител на RDBMS сървърите. Ще разгледаме неговите основни компоненти и инструменти за управление. Ще обърнем внимание на използваното от него разширение на езика SQL, наречено T-SQL и ще направим преглед на основните DDL, DML и DBCC команди. Ще разгледаме съхранените процедури в SQL Server и как той поддържа някои важни характеристики на една релационна база от данни, като транзакции, нива на изолация и др.

Релационни бази от данни

База от данни се нарича всяка организирана колекция от данни. Няма значение дали тази колекция се съхранява на хартиен носител или в паметта на компютъра – това е база от данни. Типичен пример за база от данни е телефонният указател.

Задачите за обработка на данни са едни от най-популярните в компютърните науки и като такива те имат дълга история. Затова преди да се задълбочим в разглеждането на **релационния модел**, нека да видим някои от предшестващите го.

Модели на базите от данни

Информационните системи работят с данни, които описват някакви обекти от реалния свят. За да се представят данните от реалния свят в паметта на компютрите, се използват модели за описание на данните. Тези модели задават унифицирани подходи и правила, по които обектите от реалния свят се пренасят като структури в паметта на компютъра или във външни системи за съхранение и обработка на данни.

Процесът на пренасяне на информация за обекти от реалния свят в структури от света на компютрите се нарича **моделиране на данните**. За целите на това моделиране са разработени няколко различни концепции (модели) за представяне на данните. Всеки от тях си има свои предимства и недостатъци и би могъл да се използва с различен успех при различни ситуации. Нека разгледаме най-разпространените модели на данните.

Йерархичен (дървовиден)

При йерархичния модел представянето на данните може да се опише с помощта на дърво. Тук всеки елемент има един родител и няколко наследника. Един от елементите играе ролята на корен в дървото – той има само наследници и няма родител. Именно от него започва търсенето на дадена информация. Всеки елемент си има някаква структура - списък от атрибути, които могат да имат стойности или поддървета или и двете. Пример за такава база от данни е Microsoft Active Directory, Windows Registry или файловата система на Windows например.

Този подход при организация на базите от данни се е ползвал широко в миналото, но вече намира по-рядко приложение в съвременните информационни системи.

Мрежови

Мрежовият модел представлява обобщение на дървовидния, при което базата от данни не се представя чрез дърво, а с помощта на ориентиран граф. При изграждането на графа се допуска и наличието на цикли. Пример за база от данни, работеща на този принцип, може да бъде информа-

ционната система World Wide Web (WWW) – в нея имаме страници с информация и връзки между тях.

Релационен (табличен)

Релационният модел на базите от данни е разработен от д-р Едгар Ф. Код. В своя труд озаглавен "A Relational Model of Data for Large Shared Databanks", той дефинира 13 правила, които определят една база от данни като релационна. Моделът се базира на следните дялове от математиката – теория на множествата, предикатна логика от първи ред и релационна алгебра. Терминът релация, който е част от теорията на множествата дава и името на модела.

При релационния модел, данните се съхраняват в таблици, като е възможно между отделните таблици да се задават релации (връзки). Всяка таблица е съставена от записи, които представляват редовете на таблицата. Записите се състоят от полета (клетки от таблицата), които са най-малкото количество информация, което може да бъде манипулирано в дадена релационна база от данни.

Обектно-релационен

За обектно-релационният модел може да се мисли като за хибрид между релационния модел на бази от данни и обектно-ориентираното програмиране. Към релационния модел се въвеждат някой от концепциите на обектно-ориентираното програмиране. Например при този модел таблиците могат да се наследяват за добавяне на допълнителни полета.

Системи за управление на БД

Система за управление на релационна база от данни (Relational Database Management System - RDBMS) се нарича софтуер, който осигурява:

- създаване, промяна и изтриване на таблици и връзки между тях
- добавяне, промяна, изтриване, търсене и извличане на данни от таблиците
- поддръжка на езика SQL (езикът SQL ще бъде представен по-късно в настоящата тема)
- управление на транзакциите. Транзакциите са група от промени по базата от данни, които изцяло се изпълняват или изцяло се отказват. Управлението на транзакциите не е задължителен елемент на СУБД, но се предлага като стандартна възможност от съвременните системи. По-голямо внимание на транзакциите ще обърнем по-нататък в настоящата тема.

По-разпространени RDBMS сървъри

Много често RDBMS системите се наричат още "сървъри за управление на бази от данни (СУБД)" или просто "Database сървъри". По-известни такива RDBMS сървъри са:

- **Microsoft SQL Server** (http://www.microsoft.com/sql/default.mspx). Характерно за него е лесната администрация, подобно на останалите сървъри на Microsoft, добра производителност, висока скалируемост и висока надеждност (в последните версии). Сървърът поддържа всички важни характеристики на съвременните RDBMS системи. За съжаление Microsoft SQL Server работи само под операционната система Windows. Той е комерсиален продукт и е предназначен най-вече за корпоративни клиенти. При малки проекти може да се използва неговата безплатна ограничена версия (MS SQL Server Desktop Engine).
- **Oracle Database** (http://www.oracle.com/database/index.html). Това е един от лидерите при RDBMS системите, за който е характерна изключително висока надеждност, способност за работа с голям брой потребители при огромно натоварване, възможност за управление на огромни обеми данни и версии както за Windows така и за Linux и други ОС. Представлява комерсиален продукт, който обаче може да се изтегли безплатно от сайта на Oracle за учебни цели.
- **IBM DB2** (http://www-306.ibm.com/software/data/db2/). Като водещ RDBMS сървър, отговарящ на нуждите на големите корпоративни клиенти, той е надежден, способен е да поема огромно натоварване и се скалира добре при голям брой клиенти. Комерсиален продукт. Има възможност за работа върху различни платформи.
- **PostreSQL** (http://www.postgresql.org/). Един от най-сериозните RDBMS сървъри с отворен код, който притежава висока надеждност. Базиран е на код, който първоначално е разработен от University of California в департамента по компютърни науки – Berkeley. По архитектура и организация много прилича на Oracle. Поддържа всички по-важни характеристики на RDBMS сървърите (съхранени процедури, транзакции и др.).
- **MySQL** (http://www.mysql.com/). Този сървър има славата на много бърз и лесен за използване. Той е разработен като проект с отворен код и еволюира постепенно от проста система за съхранение на данни към сериозен RDBMS сървър. В последните си версии поддържа транзакции и съхранени процедури. Заради недостатъчната си надеждност и проблемите при високо натоварване MySQL се използва рядко за големи и критични за бизнеса проекти. Характерни за него са версии както за Windows, така и за Linux, които могат да се ползват безплатно. Много често се комбинира с езика PHP при създаването на динамични уеб сайтове.
- **Borland Interbase** (http://www.borland.com/interbase). Много лек RDBMS сървър, притежаващ всички качества на съвременните RDBMS системи (съхранени процедури, транзакции и т.н.). Сървърът работи както под Windows, така и под Linux. Съществува безплатна негова версия – **Firebird** (http://firebird.sourceforge.net/) която е с отворен код.

По-нататък в настоящата тема ще разгледаме в детайли основните характеристики на релационните бази от данни и реализацията им в Microsoft SQL Server. Спираме се на SQL Server, а не на някой от другите сървъри, защото при изграждане на решения, базирани на .NET платформата, SQL Server се ползва най-масово.

Таблици

Таблиците представляват съвкупност от стойности, подредени в редове и колони. По-долу е показана примерната таблица **PERSONS**, която съдържа информация за служители във фирми:

id	name	family	employer
1	Светлин	Наков	БАРС
2	Бранимир	Гюров	BSH
3	Мартин	Кулов	CodeAttest

За всяка таблица е характерно, че нейните редове имат еднаква структура, т.е. всеки два реда от нея имат едни и същи полета (колони). Ако при проектиране на таблиците, се получи ситуация в която се налага дадени записи от една и съща таблица да имат различна структура, това означава, че подходът при проектирането е грешен и трябва да се промени.

Колоните в една таблица имат име, което трябва да бъде уникално в рамките на таблицата и тип, който определя вида на съдържаните данни. Различните СУБД поддържат различен набор от типове, но най-често срещаните като число, символен низ и дата се поддържат стандартно от всички СУБД. Ако даден тип не се поддържа, то той може да бъде имитиран чрез друг – например булевият тип може да се представи чрез символ или число, при което 1 означава истина, а 0 – лъжа.

Схема на таблица

Схема на таблица е наредена последователност от описания на колони (име и тип). Например таблицата **PERSONS** от горния пример има следната схема:

```
PERSONS (
  id: число,
  name: символен низ,
  family: символен низ,
  employer: символен низ
)
```

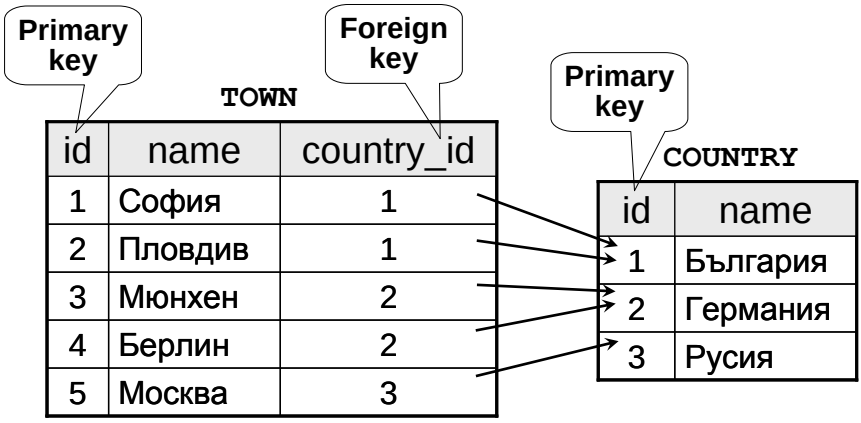
Първичен ключ

Първичният ключ (primary key) е колона от таблицата, която уникално идентифицира даден неин ред. Всички стойности в колоната, отбелязана като такъв ключ, са уникални. Затова два записа (реда) са различни тогава, когато са различни и първичните им ключове. За примерната таблица **PERSONS** първичен ключ е колоната **id**. Тя идентифицира уникално всеки един от редовете в таблицата.

Въпреки, че обикновено първичният ключ се състои от една колона, това съвсем не е задължително. Той може да е съвкупност от няколко колони.

Външен ключ

Външен ключ (foreign key) се нарича копие на първичен ключ на дадена таблица, което е включено в данните на друга таблица. Пример за външен ключ е полето **country_id** от таблицата **TOWN** показана по-долу.



Връзки (релации)

Връзка между две таблици съществува, ако може по някакъв начин да се съпоставят записи от една таблица със записи от друга таблица. Връзките се базират на взаимоотношенията primary key / foreign key. Външния ключ в една таблица (foreign key) е номер на запис (primary key) в друга таблица. В горната диаграма на таблиците, полето **country_id** от таблицата **TOWN** е външен ключ, който сочи към първичния ключ на запис от таблицата **COUNTRY**. Така двете таблици **TOWN** и **COUNTRY** са обвързани помежду си с **релация**.

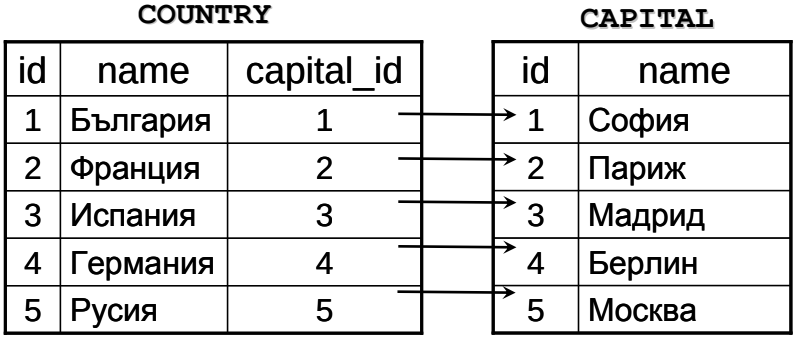
Едно от важните свойства на релациите, е че те спестяват повтарянето на информация. В горния пример името на държавата не се повтаря за всеки град, като по този начин е избегнато ненужното дублиране на информация.

Множественост на връзките

Връзките притежават така наречената **множественост** – тя показва как записите между две таблици, обхванати във връзка, се отнасят един към друг. С други думи тя показва за даден запис от една таблица, колко записа съответстват в друга таблица.

Връзка 1 x 1

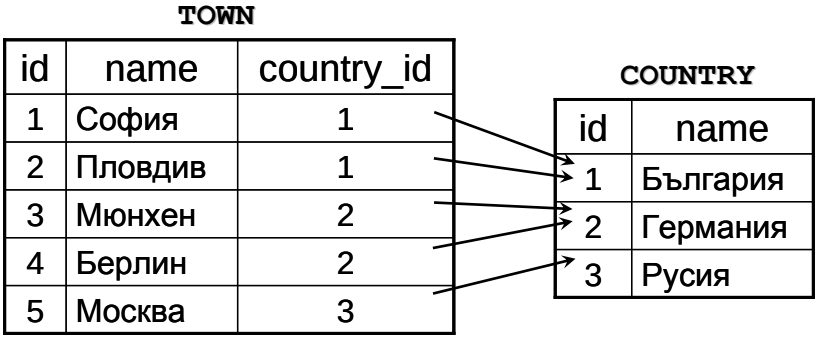
При тази връзка на един запис от първата таблица съответства точно един запис от втората таблица. Пример за такова съответствие е държава – столица. Всяка държава има една столица и всяка столица принадлежи на една държава:



Връзките от тип 1 към 1 се използват рядко, в особени случаи, и не намират широко приложение.

Връзка 1 x много (или много x 1)

При тази връзка един запис от първата таблица съответства на много записи от втората таблица. Пример за такова съответствие е град – държава. Всеки град принадлежи на една държава и всяка държава има много градове:

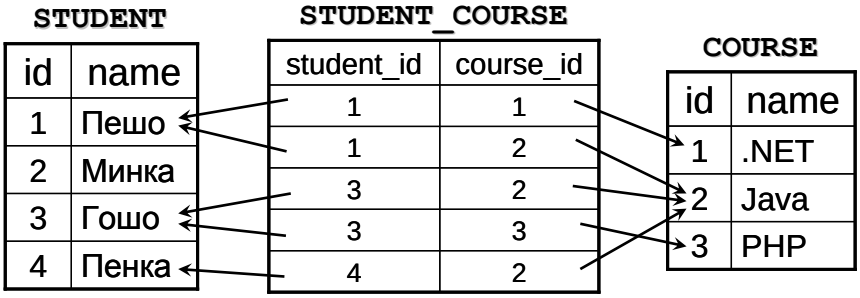


Връзките от тип 1 към много се използват постоянно при моделиране на взаимоотношения от реалния свят, където се срещат непрекъснато.

Връзка много х много

При тази връзка на един запис от първата таблица съответстват много записи от втората таблица и обратно. Пример за такова съответствие е студент – учебна дисциплина. Всеки студент изучава много учебни дисциплини и всяка учебна дисциплина е изучавана от много студенти.

Връзките от тип много към много се реализират чрез въвеждането на трета междинна таблица. Тя съдържа само две полета, които се формират от първичните ключове на двете таблици, които участват във връзката. Така поотделно всяко от полетата на свързващата таблица е и неин външен ключ, докато заедно те образуват сложен първичен ключ. Всичко това се илюстрира от дадената по-долу диаграма:



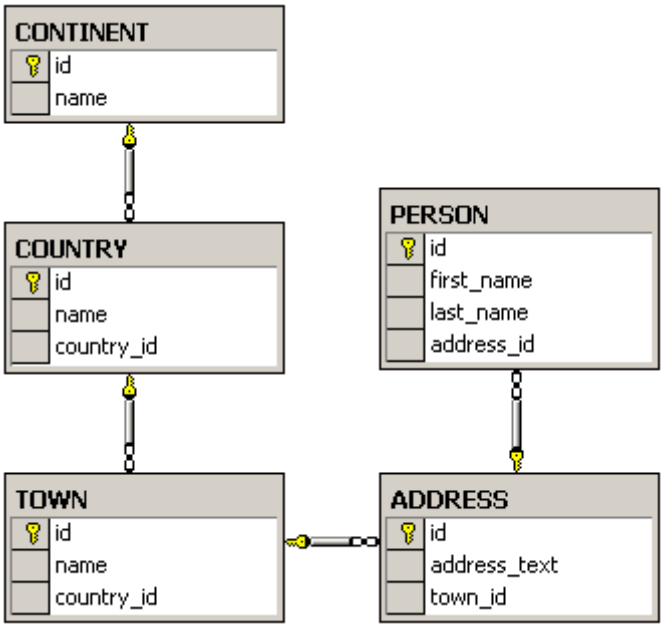
От диаграмата лесно се вижда, например, че студентът Пешо е записан в курсовете .NET и Java, а в курса Java са записани студентите Пешо, Гошо и Пенка.

Релационна схема

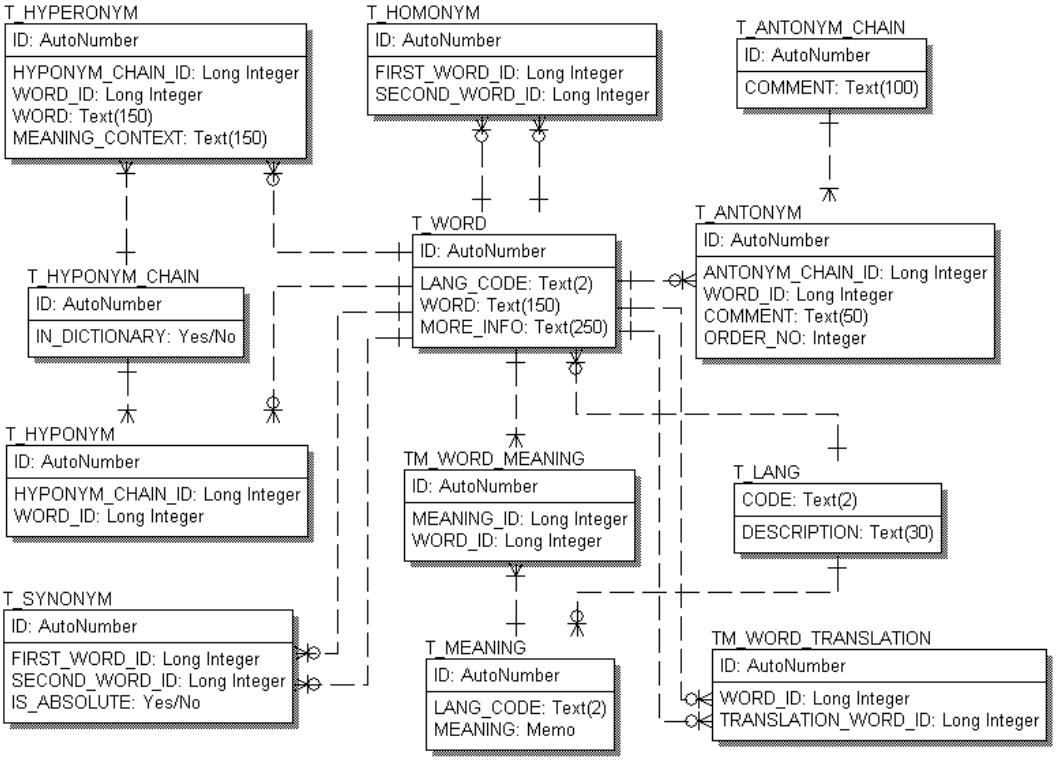
Релационна схема на БД наричаме съвкупността от схемите на всички таблици и връзките между таблиците. Тя описва структурата на БД, но не съдържа данни, а само метаданни. Метаданните са такъв тип данни, който описва други данни – например метаданните за една таблица съдържат имената и типовете на колоните и др. нейни характеристики.

E/R диаграми

Релационните схеми се изобразяват графично, чрез Entity/Relationship диаграми (диаграми "същност-връзка") или както по-често се отбелязват – E/R диаграми. Пример за такава диаграма, създадена чрез SQL Server Enterprise Manager, е дадена по-долу:



Ето още един пример. Следващата E/R диаграма е създадена чрез инструмента PLATINUM ERwin:



Връзките между таблиците на показаната по-горе диаграма са изобразени чрез линии, при които се използват специални означения за това как се отнасят записите на едната таблица към записите на другата таблица. Примерна такава връзка е показана на картинката по-долу:

В случая връзката показва, че един запис от таблицата в лявата част на връзката (задава се от първата вертикална черта), съответства на много записи от таблицата в дясната част на връзката (задава се от трите черти в десния край). Наличието на запис в лявата таблица е задължително (определено от втората вертикална черта), докато наличието на записите в дясната таблица не е задължително (определя се от кръгчето).

Ако трите черти от дясната страна бяха заменени с една вертикална, то връзката щеше да бъде от тип едно към едно. Обратното също е вярно – замяната на първата вертикална черта с три подобни на тези от дясно показва връзка от тип много към много.

Инструменти за E/R дизайн

E/R диаграмите се създават визуално, чрез инструменти за моделиране на данни. Някой от най-често използваните такива инструменти са:

- **Microsoft Visio** (http://office.microsoft.com/visio/). Освен за дизайн на E/R диаграми, може да се използва за създаване на диаграми на класове, блокови диаграми и др. При използването му като средство за моделиране структурата на база от данни има възможност за автоматично генериране на SQL скрипт, създаващ релационната схема. Друга негова характеристика е т. нар. Reverse Engineering. Това представлява възможността да се създаде диаграма на базата от данни след извличане на нейната релационна схема.
- **Oracle Designer** (http://www.oracle.com/technology/products/designer/) е специализиран инструмент за моделиране на информационни системи на базата на сървъра Oracle и свързаните с него технологии. В частност поддържа проектиране на E/R диаграми, генериране на схеми и Reverse Engineering.
- **Computer Associates Erwin** (http://www3.ca.com/Solutions/Product.asp?ID=260) е един от най-силните инструменти за дизайн на E/R диаграми. Той работи с много различни видове релационни бази от данни, предлага възможност за автоматично генериране на SQL скрипт, както и възможността за Reverse Engineering.
- **SQL Server Enterprise Manager** (http://www.microsoft.com/sql/). Представлява стандартен административен инструмент, който се доставя като част от Microsoft SQL Server. Една от възможностите, които този продукт предлага, е дизайн на E/R диаграми. Естествено, създадените диаграми могат да се използват само с MS SQL Server.

- **IBM Rational Rose Data Modeler** (http://www-306.ibm.com/software/awdtools/developer/datamodeler/). **Rational Rose** представлява огромен пакет от инструменти за моделиране на софтуерни системи по модела "Rational Unified Process". **Rational Rose Data Modeler** предлага средства за моделиране на бази от данни. Поддържа E/R диаграми, генериране на SQL скриптове и Reverse Engineering.
- **theKompany Data Architect** (http://www.thekompany.com/products/dataarchitect/) е инструмент за моделиране на E/R диаграми, който има версии за Windows, Linux, Mac OS X. Работи с различни RDBMS сървъри.
- **fabForce DBDesigner** (http://www.fabforce.net/dbdesigner4/) е GNU GPL проект с отворен код за Windows и Linux. Позволява моделиране на данни чрез E/R диаграми и поддържа различни RDBMS сървъри, като специално е оптимизиран за работа с MySQL.

Нормализация

Нормализация се нарича процесът на привеждане структурата на една база от данни в съответствие с правилата за нормализация. Причината, поради която трябва да извършваме нормализация на нашата база от данни, е съдържанието на многократно повтарящи се данни в таблиците. В тази тема ще разгледаме 4 от правилата за нормализация и как те подобряват релационната схема на нашата база от данни. По-долу е дадена таблица, която съдържа множество повтарящи се данни и не отговаря на правилата за нормализация:

продукт	производител	цена	категория	магазин	град
кисело мляко	Млекис ООД	0.67	хранителни стоки	супермаркет "Менте"	София
хляб "Добружда"	Фурна "Пушека"	0.55	хранителни стоки	супермаркет "Менте"	София
бира "Загорка"	Загорка АД	0.58	безалкох. напитки	павилион "24 часа"	Варна
бира "Tuborg"	Шуменско пиво АД	0.67	безалкох. напитки	павилион "24 часа"	Варна

1-ва нормална форма

Една релационна база от данни може да се смята, че е в първа нормална форма, когато са изпълнени следните изисквания:

- данните имат табличен вид
- полетата в редовете са с атомарни (неделими) стойности, т.е. всяко поле съдържа само една стойност, а не списък от стойности

- няма повторение на данни в рамките на един ред
- дефиниран е първичен ключ за всяка таблица

Дадената по-долу таблица отговаря на критериите за първа нормална форма. Данните са оформени в табличен вид, има дефиниран първичен ключ (ISBN), колоните са с неделими стойности и няма повторение на данните в рамките на един ред.

книга	**ISBN (PK)**	**автор**	**автор_email**
.NET Framework	3847028437	Бай Киро	bai-kiro@abv.bg
Beginning SQL	7234534450	Дядо Мраз	dedo@mraz.org

В горния пример би било грешно да се обединят колоните "книга" и "ISBN", защото така ще се наруши правилото всяко поле да съдържа неделими стойности.

2-ра нормална форма

Една таблица е във втора нормална форма, когато тя се намира в първа нормална форма и всяко нейно поле зависи от всички части на първичния ключ за съответния ред (ако той се състои от няколко колони).

Дадената по-долу примерна таблица не отговаря на изискванията за втора нормална форма, защото колоната "цена" зависи единствено от колоната "книга", която е част от първичния ключ (който в случая се състои от колоните "книга" и "автор"). Същото се отнася и за колоната "автор_email", която зависи единствено от колоната "автор".

книга (PK)	**автор (PK)**	**цена**	**автор_email**
.NET Framework	Бай Киро	27.25	bai-kiro@abv.bg
Beginning SQL	Дядо Мраз	19.95	dedo@mraz.org
Introduction to Delphi	Бай Киро	20.56	bai-kiro@abv.bg

За да се приведе таблицата във втора нормална форма, ще трябва да се разбие на три отделни таблици, които имат следната структура:

книга (PK)	**автор (PK)**
.NET Framework	Бай Киро
Beginning SQL	Дядо Мраз
Introduction to Delphi	Бай Киро

книга (PK)	цена
.NET Framework	27.25
Beginning SQL	19.95
Introduction to Delphi	20.56

автор (PK)	автор_email
Бай Киро	bai-kiro@abv.bg
Дядо Мраз	dedo@mraz.org

3-та нормална форма

Една таблица е в трета нормална форма, ако тя се намира във втора нормална форма и всичките ѝ полета, които не са първични или външни ключове, са взаимно независими. С други думи, нейните полета зависят само от първичния ключ.

Ето един пример за таблица, която не отговаря на тези условия:

id	продукт	производител_id	цена	категория
1	кисело мляко	2	0.67	хранителни
2	хляб "Добруджа"	3	0.55	хранителни
3	ракия "Пещерска"	6	4.38	алкохол
4	бира "Tuborg"	4	0.67	бира

В примерната таблица полето "категория" зависи от полето "продукт".

Следващата таблица вече е нормализирана до 3-та нормална форма. При нея неключовите полета "продукт" и "цена" зависят единствено от първичния ключ, чиято роля се изпълнява от полето "id":

id	продукт	производител_id	цена	категория_id
1	кисело мляко	2	0.67	2
2	хляб "Добруджа"	3	0.55	2
3	ракия "Пещерска"	6	4.38	5
4	бира "Tuborg"	4	0.67	4

4-та нормална форма

Една таблица се намира в четвърта нормална форма, ако тя се намира в трета нормална форма и в таблицата има най-много една колона с няколко възможни стойности за един ключ.

Дадената по-долу примерна таблица не отговаря на изискванията за четвърта нормална форма, тъй като всяка от колоните "книга" и "статия" съдържат атрибути с няколко възможни стойности за един и същ ключ (автор_id), а това противоречи на дадената по-горе дефиниция за четвърта нормална форма:

автор_id	книга	статия
2	.NET Programming	Regular Expressions in .NET
4	Mastering J2EE	Best Practices in J2EE

За да се удовлетворят изискванията на четвърта нормална форма за горната таблица, колоната статия би трябвало да се премести в нова таблица, която да съдържа следните колони: "автор_id" и "статия".

4-та нормална форма – пример

Един по-завършен пример за база от данни, която е нормализирана до четвърта нормална форма е даден на следващата фигура:

Предимства на нормализираната база от данни

Една нормализирана база от данни има сериозни предимства пред такава, в която не са спазени изброените по-горе правила. За пример нека да вземем база от данни, която моделира телефонен указател. Ако име, презиме и фамилия са представени чрез едно поле (т.е. не е спазено първото от правилата за нормализация), то търсенето в подобна база от данни по фамилия ще бъде затруднено.

При дадената денормализирана примерна таблица в точката, обясняваща втора нормална форма, е проблемна смяната на електронния адрес на автора. Ако Бай Киро иска да си смени адреса от bai-kiro@abv.bg на kiro-menteto@abv.bg, то това ще трябва да бъде извършено в няколко реда на таблицата. Както знаем промяната на няколко реда в една таблица е значително по-бавна операция от промяната на един ред, но това не е най-страшното. По-големият проблем е, че има риск да се обновят само някои от записите и така да се наруши интегритета на данните в табл-

ицата – Бай Киро да има няколко e-mail адреса – някои от които валидни, а други – не.

Използването на денормализирани бази от данни е оправдано в така наречените **OLAP** (On-Line Analytical Processing) системи. При тях се извършва предимно извличане на информация от базата от данни и някой таблици нарочно се поддържат денормализирани с цел по-бързото генериране на нужния набор от данни.

Ограничения (Constraints)

Ограниченията задават правила за данните, които не могат да бъдат нарушавани. RDBMS сървърите дават грешка при опит за промяна, която нарушава някое ограничение. Те забавят работата на сървъра, но спомагат за интегритета на данните. Ограниченията биват няколко вида:

- **Ограничения по първичен ключ** (primary key constraint) – първичният ключ във всяка една таблица е винаги уникален.
- **Ограничения по уникален ключ** (unique key constraint) – стойността в дадена колона или група колони е уникална, т.е. стойността на полето за всеки ред от таблицата е уникална в рамките на самата таблица.
- **Ограничение по външен ключ** (foreign key constraint) – това ограничение налага правилото, че стойността в дадена колона е ключ от друга таблица.
- **Ограничение по стойност** (check constraint) – това ограничение налага правилото, че стойността в дадена колона изпълнява някакво предварително зададено условие. Например за колоната **`hour`** може да имаме ограничението: **`(hour >= 0) AND (hour <= 24)`**.

Индекси

Индексите представляват допълнителна информация за дадена таблица, която ускорява процеса на търсена на стойност в дадена колона или група колони. Най-често се реализират с B-дървета или хеш-таблици. Естествено реализацията е оставена на СУБД, като потребителят се възползва наготово от тях. Обикновено индексите се ползват при големи таблици, които често се претърсват за определени данни.

Предимството на индексите в процеса на търсенето е за сметка на модифицирането на данните. Наличието на индекс за дадена таблица забавя добавянето и изтриването на записи от нея заради нуждата от поддръжка на индекса.

Различните видове индекси са подходящи в различни ситуации. Например, ако често се търси определена стойност в дадена колона, хеш-индексите работят много добре, но ако често се търсят всички стойности в даден интервал от дадена колона, трябва да се използва индекс B-дърво.

Езикът SQL

Езикът **SQL** (Structured Query Language) представлява стандартизиран език за манипулация на релационни бази от данни. В момента има няколко стандарта, но най-разпространените от тях са два – SQL-92, който се поддържа от всички релационни бази от данни и SQL-99, който все повече навлиза в употреба.

Data Definition Language (DDL)

Като възможности езикът SQL предлага конструкции за създаване, промяна, изтриване на таблици и други обекти в базата от данни. Тази част от езика се нарича **DDL** (Data Definition Language) и обхваща команди като **`CREATE`**, **`ALTER`**, **`DROP`**.

Data Manipulation Language (DML)

Другата част от езика е наречена **DML** (Data Manipulation Language) и предлага команди за търсене извличане, добавяне и изтриване на данни. Към нея спадат команди като **`SELECT`**, **`INSERT`**, **`UPDATE`** и **`DELETE`**. Прост пример за SQL команда, която извлича от таблицата **`People`** всички имена на хора, чиято фамилия е **`Иванов`**, е показана по-долу:

```
SELECT FirstName, LastName
FROM People
WHERE LastName = 'Иванов'
```

Освен DDL и DML командите, много RDBMS сървъри поддържат специфични разширения на езика. Едни от по-известните разширения на езика SQL са T-SQL, поддържан в Microsoft SQL Server, и PL/SQL, използван в Oracle и PostgreSQL.

Изгледи (Views)

Изгледите представляват виртуални таблици, които се състоят от полета от една или повече физически таблици. Изгледите са виртуални, защото те не съхраняват данни – само показват по друг начин вече съществуващите данни от таблиците.

Изгледите се използват обикновено за улесняване писането на сложни SQL заявки. Те скриват сложната заявка, като позволяват да се работи с върнатите от нея данни по начин, все едно тези данни идват от реална, физическа таблица. Така всеки, който иска да ползва резултата от сложната заявка, може да го достъпи просто чрез името на изгледа.

Друго характерно приложение на изгледите е при задаване на фина настройка на сигурността. Например нека имаме таблица до която искаме да дадем достъп на потребителите само върху част от колоните ѝ. Тогава създаваме изглед, който съдържа само колоните от таблицата, върху които искаме да дадем достъп и разрешаваме всички потребители в системата да го ползват. Същевременно премахваме достъпа на потреби-

телите до въпросната таблица. По този начин чрез използването на изгледи даваме достъп на потребителите до подмножество от данните.

Пример за изглед от няколко таблици

Да разгледаме следната ситуация: Дадени са множество държави, градове и фирми. В една държава може да има много градове, а във всеки град може да има регистрирани много фирми. Така всяка фирма принадлежи на някой от градовете и респективно на някоя от държавите.

Можем да моделираме тази ситуация с три таблици по следния начин:

T_COMPANY

id	company	town_id
1	Менте ООД	1
2	BulkSoft Inc.	2
3	ХардСофт АД	1
4	Спутник АД	3

T_TOWN

id	town	country_id
1	София	1
2	New York	3
3	Москва	2

T_COUNTRY

id	country
1	България
2	Русия
3	САЩ

Ако разработваме информационна система за обслужване на фирмите по държави, може да се наложи често пъти да извличаме и обработваме всички български фирми. За улеснение на работата можем да дефинираме следния изглед с име **V_BG_COMPANY**, който комбинира данни от таблиците **T_COMPANY**, **T_TOWN** и **T_COUNTRY**:

```
CREATE VIEW V_BG_COMPANY AS
  SELECT
    T_COMPANY.id AS id,
    T_COMPANY.company AS company
  FROM T_COMPANY INNER JOIN
    (T_TOWN INNER JOIN T_COUNTRY ON
    T_TOWN.country_id=T_COUNTRY.id)
    ON T_COMPANY.town_id=T_TOWN.id
  WHERE
    T_COUNTRY.country="България";
```

Дефиницията на изгледа съединява (INNER JOIN) трите таблици по съответните им връзки първичен-външен ключ и филтрира получените записи така, че да останат само тези от държавата "България" (различните видо-

ве съединения на таблици, като "вътрешно съединение" ще бъдат обяснени по-късно в настоящата тема).

Резултатът, който ще се получи при опит за извличане на всички редове от дефинирания изглед **V_BG_COMPANY** е следният:

id	company
1	Менте ООД
3	ХардСофт АД

Съхранени процедури (stored procedures)

Съхранените процедури (процедури на ниво база, запазени процедури) представляват програмен код, състоящ се от последователност от SQL команди, които се изпълняват в самия сървър за бази от данни. Когато дадена клиентска машина иска да изпълни дадена съхранена процедура, тя просто подава името на процедурата, а сървърът изпълнява всички SQL команди от процедурата.

Горното обяснение всъщност е малко опростено с цел по-лесно разбиране. Реално съхранените процедури на сървъра, подобно на процедурите в останалите езици за програмиране могат да приемат параметри и да връщат резултат. Връщаният резултат може да бъде както единична стойност, така и съвкупност от записи (record set).

Освен SQL команди съхранените процедури могат да ползват и логически конструкции, цикли, изключения и други характерни за езиците от високо ниво конструкции.

Съхранените процедури се пишат на разширенията на езика SQL. Пример за съхранена процедура на сървъра, написана на T-SQL, която връща име на продукт и количество в зависимост от име на склад, подадено като параметър, е даден по-долу:

```
CREATE PROCEDURE spGetInventory
   @location varchar(10)
AS
   SELECT Product, Quantity
   FROM Inventory
   WHERE Warehouse = @location
```

Предимства на съхранените процедури

Използването на съхранени процедури има няколко предимства пред използването на обикновени SQL команди.

При създаването на нова процедура, сървърът прави синтактичен анализ на съставящите я SQL команди и оптимизира тяхното изпълнение, така че то да бъде най-ефективно. Тези действия се извършват еднократно (процесът се нарича компилация), докато при обикновените SQL команди,

това става всеки път при постъпване на командата за изпълнението ѝ. Разбира се, ако SQL командата е била изпълнявана преди това, тя не се компилира наново, защото вече се намира в кеша с командите на сървъра. За да се случи това реално са необходими някой допълнителни условия като използване на параметризирани SQL заявки. Пример за такива ще дадем при разглеждане на библиотеката от класове ADO.NET в темата "Достъп до данни с ADO.NET".

Друго преимущество на съхранените процедурите, е че те могат да реализират сложен алгоритъм за обработка на информацията от страна на сървъра, без да има нужда тя да се прехвърля до клиентската машина. По този начин се намалява обема на информация обменян между сървъра и клиента и в резултат на това, се намалява мрежовия трафик.

Представете си за пример, че в една банка трябва да се извърши олихвяване върху 10 000 000 сметки по някаква сложна формула. Ако олихвяването на един запис изисква неговото извличане от сървъра към клиента, пресмятане на лихвата и последващо обновяване на записа в базата данни, това ще предизвика огромен мрежов трафик и ще намали значително производителността. В такъв случай е силно препоръчително да се използва съхранена процедура, която извършва олихвяването на сървъра.

Тригери (Triggers)

Тригерите представляват програмен код, който се изпълнява автоматично при настъпване на някакво събитие в базата от данни. Примери за такива събития са:

- добавяне на запис в таблица
- промяна на запис в таблица
- изтриване на запис в таблица

Изпълнението на тригерите се извършва на сървъра, подобно на съхранените процедури. Тригерите се пишат на T-SQL, PL/SQL или друго разширение на SQL, като те могат да извикват съхранени процедури.

Тригерите обикновено се използват за извършване на допълнителна обработка на данните, например при добавяне на запис или за поддръжка на логове и история.

Пример за тригер

Ето един пример за тригер, който при добавяне на нова фирма в таблицата **COMPANY** слага "**Ltd.**" в края на името ѝ:

```
CREATE TABLE COMPANY(
   id int NOT NULL,
   name varchar(50) NOT NULL)

CREATE TRIGGER trg_COMPANY_INSERT
```

```
   ON COMPANY
   FOR INSERT
AS
   UPDATE COMPANY SET name = name + ' Ltd.'
   WHERE id = (SELECT id FROM inserted)
```

Показаният по-горе тригер се активира при добавяне на нов запис към таблицата **COMPANY**. Той изпълнява SQL заявка, която обновява току-що добавения запис, като добавя " **Ltd**." в края на полето за име. Идентификаторът на полето, който трябва да бъде обновен, се намира чрез извличане от таблицата **inserted**. Това е специална таблица в MS SQL Server, в която се пазят всички записи, които са били засегнати в резултат от последната **INSERT** или **UPDATE** заявка.

Транзакции

Транзакциите представляват последователности от действия (заявки към базата от данни), които се изпълняват атомарно. При транзакциите или всички действия обхванати в транзакцията се изпълняват успешно или никое от тях не се изпълнява изобщо. Използването на транзакции е наложително, когато една операция, свързана с модифициране на базата от данни, се състои от две или повече стъпки (например при промяна на повече от една таблица).

Типичен пример за приложение на транзакциите е сценарият с прехвърлянето на пари от една банкова сметка в друга. Това действие е свързано с извършването на две операции – теглене на сумата за прехвърляне от първата сметка и внасянето ѝ към втората. Двете операции или трябва да бъдат изпълнени заедно или никоя от тях да не бъде изпълнена. Ако например тегленето успее, а внасянето не, клиентът ще е загубил съответната сума пари, което в една банка е недопустимо. Затова при използването на транзакции ако тегленето или внасянето на парите пропадне, пропада и цялата операция.

Пример за транзакция

Ето един пример за транзакция, която прехвърля зададена сума пари от една сметка в друга:

```
CREATE TABLE ACCOUNT(
   id int NOT NULL,
   balance decimal NOT NULL)

CREATE PROCEDURE sp_Transfer_Money(
   @from_acc int,
   @to_acc int,
   @ammount decimal
) AS
BEGIN TRANSACTION
```

```
UPDATE ACCOUNT set balance = balance - @ammount
WHERE id = @from_acc

IF @@rowcount <> 1 BEGIN
   ROLLBACK TRANSACTION
   RAISERROR ('Invalid source account!', 16, 1)
   RETURN
END

UPDATE ACCOUNT set balance = balance + @ammount
WHERE id = @to_acc

IF @@rowcount <> 1 BEGIN
   ROLLBACK TRANSACTION
   RAISERROR ('Invalid destination account!', 16, 1)
   RETURN
END

COMMIT TRANSACTION
```

Как работи примерът?

Прехвърлянето се извършва чрез процедура на сървъра, написана на T-SQL, която приема три параметъра – номер на сметка-източник, номер на сметка-получател и сума на превода.

За да се гарантира целостта на данните в началото на съхранената процедура се стартира транзакция. След това се намалява сумата за превод от сметката източник. Чрез системната променлива **@@rowcount**, която връща броя на засегнатите записи от последната операция, се прави проверка дали операцията е била успешна. Ако това не е така (например защото сметката не съществува или нямаме достъп до нея), се отказва текущата транзакция и се връща съобщението "**Invalid source account**", след което изпълнението на съхранената процедурата се прекратява.

Ако намаляването на сумата за превод от сметката източник успее, се пристъпва към увеличаване на наличността в сметката получател. В този момент отново се прави проверка по аналогичен начин за това дали операцията е минала успешно. Ако поради някаква причина възникне грешка, се отказва текущата транзакция, връща се съобщението за грешка "**Invalid destination account**", и изпълнението на съхранената процедура се прекратява.

При отказ на текущата транзакция състоянието на базата данни се връща такова каквото е било в началото на изпълнението на съхранена процедура, т.е. промените по таблицата **ACCOUNT** се анулират. Това автоматично връщане на частичните промени в рамките на транзакцията се осигурява от сървъра (от неговия **transaction manager**).

Ако увеличаването на сумата за превод в сметката получател успее, текущата транзакция се потвърждава и изпълнението на съхранената процедура приключва успешно. Потвърждаването на транзакцията гарантира записването на извършените в рамките на нейната работа промени.

В крайна сметка процедурата или успява успешно да прехвърли парите или дава грешка и не променя баланса по никоя от сметките. Благодарение на транзакцията няма как да се получи частично прехвърляне на пари – парите или се прехвърлят или не.

Отговорности на транзакциите

Транзакциите в повечето СУБД имат 4 важни характеристики:

- **Атомарност** (atomicity) – или всички действия от транзакцията се изпълняват успешно или нито едно от тях не успява.
- **Цялост на данните** (consistency) – осигурява транзакцията да не позволи на базата данни да влезне в неправилно състояние.
- **Изолация на данните** (isolation) – при паралелна работа на няколко транзакции, те са изолирани една от друга (и не си пречат). Изолацията осигурява синхронизация при достъпа до общи данни от няколко клиента едновременно. Съществуват няколко нива на изолация, които ще разгледаме след малко.
- **Стабилност на данните** (durability) – ако една транзакция приключи успешно, то тя не може да бъде загубена. Промените които тя е направила по базата от данни остават записани (дори ако спре токът или се случи софтуерен срив, не се губят данни).

Изолация на транзакциите

За транзакциите може да се дефинират нива на изолация. Нивото на изолация на дадена транзакция определя до каква степен тя ще вижда промените по базата от данни, извършени от други паралелно изпълнявани върху нея транзакции. Нивата на изолация могат да бъдат различни при различните RDBMS сървъри, но стандартните 4 нива са следните:

- **Read uncommitted**. При това ниво дадена транзакция може да прочете модифицирани данни от друга транзакция, преди втората да е завършила. Този режим на работа е опасен, защото ако втората транзакция се провали и откаже направените промените, текущата транзакция ще работи с грешни данни.
- **Read committed**. Транзакциите работещи с това ниво на изолация не могат да четат данни, които са били модифицирани от други транзакции, които не са приключили до този момент. При опит за това, командата за четене блокира докато другата транзакция не приключи.
- **Repeatable read**. В това ниво на изолация, за разлика от предното се гарантира, че ако една транзакция прочете един запис, то той ще

бъде същият докато тя не приключи работата си. Нека си представим ситуация, в която дадена транзакция прочита запис от базата от данни. След като тя е прочела данните, друга транзакция модифицира така прочетения запис и приключва работата си. Това означава, че първата транзакция работи с неверни данни, защото съдържанието на базата от данни се е променило. Именно от такива ситуации ни предпазва това ниво на изолация. При repeatable read изолация се използва заключване на записите, които бъдат прочетени. Транзакциите блокират при опит за достъп до заключен запис докато той не бъде отново отключен.

- **Serializable**. Транзакциите, работещи с това ниво на изолация, гарантират, че броят на редовете прочетени от дадена **SELECT** заявка върху дадена таблица ще бъде един и същ през цялото време на изпълнение на транзакцията. С други думи – тук липсват "фантомни" записи, защото транзакциите се изпълняват така сякаш работят една след друга. Наличието на фантомни записи се наблюдава, в ситуации, когато дадена паралелна на текущата транзакция добави нови записи към някоя таблица и приключи работа. Тогава при повторен опит за четене на записи, нашата транзакция, ако не работи с ниво на изолация serializable, ще прочете и добавените записи от паралелната транзакция.

Сравнение между отделните нива на изолация е направено в следващата таблица:

ниво на изолация	четене на непотвърдени данни	неповторяемост при четене	фантомни записи
Read uncommitted	да	да	да
Read committed	не	да	да
Repeatable read	не	не	да
Serializable	не	не	не

По-високите нива осигуряват по-добра консистентност на данните, но работят по-бавно и заключват данните за по-дълго време. Затова нивото на изолация трябва внимателно да се подбира в зависимост от конкретната ситуация.

Управление на транзакциите

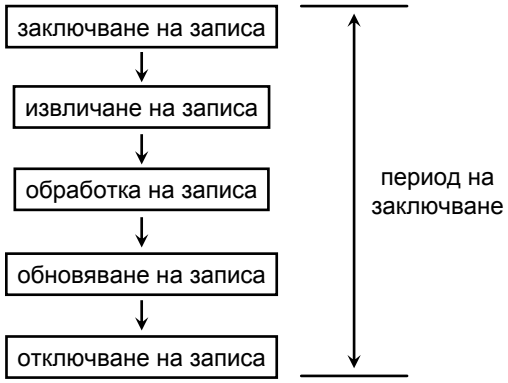
Управлението на транзакциите се извършва от т. нар. **менажер на транзакции** (transaction manager). При повечето реализации той записва всички транзакции в специална структура, т. нар. **transaction log**. Записаната там информация се използва при срив в системата и в някои други ситуации, например за повтаряне на изгубена транзакция, при репликация и т.н.

Взаимодействайки си с **мениджъра на заключванията** (lock manager), менажерът на транзакциите може да заключи определени редове, страници (няколко реда от таблица) или дори цели таблици. Най-често заключванията се реализират на ниво ред (запис) в дадена таблица.

Има два основни модела на заключване при конкурентната обработката на данни управление с транзакции – песимистично и оптимистично.

Песимистично заключване

При песимистично заключване даден ред от таблица се заключва в момента на извличането му от дадена таблица, след това се обработва от приложението и се отключва в момента на обновяването му в базата данни. През цялото време, през което записът се обработва, той стои заключен. През този период всички транзакции, които искат да работят конкурентно със заключения запис, трябва да изчакат докато той бъде освободен. Схематично можем да изобразим песимистичното заключване по следния начин:



Песимистично заключване се използва най-вече в два сценария:

- При записи, които се променят изключително често, и при които цената на заключването (и породеното от него забавяне на всички транзакции, чакащи за заключения запис) е много по-ниска, отколкото евентуално отказване на транзакция заради настъпила междувременно промяна.
- При системи, в които е пагубно даден запис да бъде променен междувременно докато се обработва в хода на изпълнение на дадена транзакция.

Песимистичния модел на заключване при обработката на данните може да доведе до сериозни проблеми с производителността при голямо натоварване, особено ако се изпълняват голям брой конкурентни транзакции.

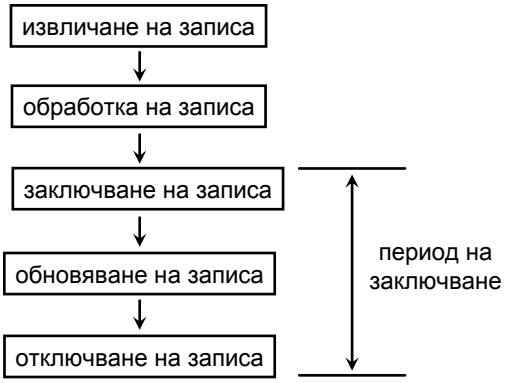
Възможно е дори настъпване на "мъртва хватка" (dead lock) – ситуация, в която една транзакция изчаква освобождаването на запис, заключен от друга транзакция, а тя съответно изчаква първата да освободи някой друг запис. За избягването на такива ситуации трябва внимателно да се анализират сценариите за паралелно изпълнение на транзакциите и да се

проектира работата с общите записи така, че да не настъпват конфликтни заключвания.

Песимистично заключване не трябва да се прилага при продължителни транзакции, защото може да причини прекалено дълго (дори безкрайно) чакане за даден заключен запис. Например при уеб приложения не можем да караме даден потребител да чака за даден запис докато друг потребител е заредил в своя уеб браузър същия запис и го редактира. В такива ситуации се използва оптимистично заключване.

Оптимистично заключване

При оптимистичното заключване даден запис се извлича, след което се обработва без да се заключва. След като обработката приключи, записът се заключва и се нанасят промените по него. Реално заключването трае само за периода, в който се нанасят промените. Заключването е необходимо, защото може да се получи конфликт ако две транзакции обновяват в даден момент един и същ запис. Схематично процесът можем да изобразим по следния начин:



Разбира се, има риск в периода между извличането на записа и нанасяне на промените в базата данни той да бъде обновен от друга транзакция. В този момент се получава конфликт, който трябва да се реши по някакъв начин.

Оптимистичното заключване държи заключени записите за много кратък период от време, не през цялото време на обработката им. По тази причина той е подходящ за приложения с голямо натоварване и много конкурентни заявки, както и при приложения, в които обработката на даден запис може да трае дълго (например уеб приложения).

Въведение в SQL Server

След като направихме преглед на релационните бази от данни, системите за управление на релационни бази от данни и понятията и технологиите, свързани с тях, ще разгледаме един конкретен сървър за управление на релационни бази от данни – Microsoft SQL Server.

Избрахме SQL Server, не само защото е един от водещите RDBMS сървъри на световния пазар, но и защото в света на .NET технологиите той е най-предпочитаният сървър за управление на данни. Понеже .NET Framework и MS SQL Server са разработени от един и същ производител (Microsoft), интеграцията между тях е отлична.

История на SQL Server

Първоначалните версии на SQL Server са свързани с усилията по създаването на продукта между Microsoft и Sybase. Microsoft поемат усилията за лансиране на продукта върху операционната система на IBM – OS/2, докато Sybase се заемат с работните станции SUN, работещи под Unix. След неоправданите очаквания за доминация на OS/2 на софтуерния пазар, Microsoft прехвърлят SQL Server под операционната система Windows NT, като същевременно се разваля и споразумението със Sybase за съвместна разработка. Всяка от двете компании тогава има възможността сама да развива продукта. Версията разработвана от Sybase се нарича Sybase Adaptive Server.

Първата версия, която Microsoft разработват изцяло самостоятелно, е версия 6, която е обявена в средата на 1995 г. Това е версията в която се появяват някой от ключовите елементи на сървъра, като SQL Enterprise Manager, възможностите за репликация на данни и т.н. Десет месеца по-късно компанията пуска и версия 6.5 на продукта, в която се развиват услугите предлагани от сървъра за анализ на данни (data warehousing).

Следващата версия 7.0 на продукта е обявена официално през 1998 година. Сървърът излиза с изцяло преработена архитектура, която включва изцяло пренаписана машина за бази от данни, както и чисто нов метод за управление на заключването, възстановяващи алгоритми, дневник на транзакциите и др.

През 2000 година е пусната следващата версия на продукта – Microsoft SQL Server 2000. Тя включва много нови възможности като разпределени частични изгледи (distributed partitioned views) и др. Това е версията, която е с най-широко разпространение в момента.

Както може да се и очаква, Microsoft разработват нова версия на продукта с кодовото название Yukon (SQL Server 2005), която към момента е в етап на бета версия. Тя ще включва някой значителни подобрения, като вграждане на CLR в ядрото на сървъра, като по този начин ще се предостави възможност за писане на съхранени процедури на C#.

Системни компоненти на SQL Server 2000

Системните компоненти на SQL Server 2000 могат да се разделят най-грубо на два вида – услуги, които предлага сървърът и инструменти за работа с него.

Услуги

- **MSSQLServer –** това е машината за бази от данни на Microsoft SQL Server 2000. Тя управлява всички файлове, които участват в дадена база от данни, притежавана от сървъра, обработва и изпълнява всички SQL команди, които постъпват за изпълнение. В нея се управлява сигурността, изграждат се и се използват индекси и т.н.
- **SQLServerAgent –** предлага възможности за задаване на периодично изпълнявани задачи върху сървъра или известяване за възникнали проблеми.
- **MSSQLServerADHelper –** използва се за интеграция на SQL Server с Active Directory. Тази услуга добавя или премахва обектите, които се използват за регистриране на инстанции на Microsoft SQL Server в Active Directory.
- **MSSQLServerOLAPService –** предлага инструменти за анализиране на данни, съхранени в складове за данни (data warehouse). Някой видове заявки обхващат огромно количество данни, поради което тяхното изпълнение се забавя много. Заради това обикновено данните, които се засягат от такъв вид заявки, се обобщават и съхраняват в такъв вид склад за данни.

Инструменти

- **Enterprise Manager –** това е основният инструмент, който се използва за администриране на сървъра. С негова помощ се изпълняват задачи, като създаване на нови бази, създаване и възстановяване на архивни копия на базата от данни, създаване на таблици, изгледи, диаграми, управление на индекси на таблици и др. MS SQL Server Enterprise Manager предоставя интерфейс съвместим с Microsoft Management Console (MMC).
- **Query Analyzer –** представлява инструмент за създаване, тестване и настройка на T-SQL скриптове. Той предлага шаблони за ускоряване процеса на разработка, интерактивен дебъгер, графична диаграма на плана на изпълнение на SQL заявката и др. Използва се най-често за изпълнение на SQL заявки в интерактивен режим.
- **DTS (Data Transformation Services) –** използват се при необходимост за дефиниране на решения за преместване на данни от и към външни източници. Преди данните да пристигнат до определяната им цел, има възможност да се зададе изпълнението на дадена трансформация над тях. За източници на данни могат да служат други релационни бази от данни (като например Oracle), както и източници на данни съхранявани в различен от релационния модел (например Microsoft Exchange).
- **SQL Profiler –** това е инструмент, който прихваща събития от SQL Server. Такива събития могат да бъдат например изпълнението на SQL команда, включване на потребител към сървъра и др. След като

бъдат прихванати подобни събития, информацията от тях може да бъде използвана за да се идентифицират изпълняващи се бавно SQL заявки с цел да се оптимизират, проследяване на серия от команди, водещи до проблеми и др. С други думи, това е инструментът, чрез който можете да надникнете в сърцето на SQL Server.

- **SQL XML –** предлага възможности за изпълнение на SQL заявки, които връщат XML като резултат. Дава възможности за публикуване на данни в уеб среда (в Internet Information Services - IIS).
- **Analysis Manager -** предоставя възможности за администриране на Analysis Server. Analysis Server е сървърният компонент на MSSQLServerOLAPService.

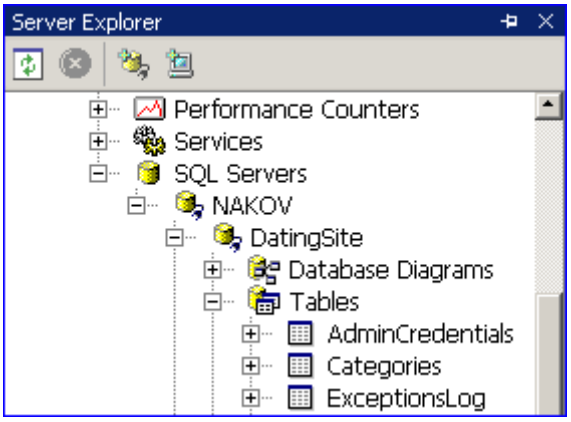
Програмиране за SQL Server 2000

Програмните среди, които се използват най-често при разработка на приложения на .NET включващи SQL Server са: Visual Studio .NET 2003 и Query Analyzer.

Visual Studio .NET

Visual Studio .NET включва някои инструменти, които спомагат разработката на приложения, свързани с бази от данни. Към тях спада и т. нар. Server Explorer. Той представлява конзола за менажиране на сървъри (включително и такива за бази от данни). Визуализирането му на екрана става чрез менюто View → Server Explorer. Чрез него можете да разглеждате и манипулирате различните обекти на една база от данни, дневника на събитията на Windows (event log), следите различни показатели на производителността и др.

Ето как изглежда Server Explorer:



Интересна възможност, която се предлага от Server Explorer е разглежданите ресурси (като таблици в дадена база от данни), които можете да манипулирате чрез него, да се привлачват в дадена форма на приложението. При тази операция автоматично се генерират съответните обекти

за достъп до таблицата. Предлага се и възможност за постъпково изпълнение на съхранени процедури на сървъра в средата на Visual Studio .NET.

Друга възможност, която предлага средата за разработка, е създаването на т. нар. проекти за бази от данни (DB Projects). Те представляват групирането на отделни скриптове, които създават обектите на базата от данни в един проект. От този проект има възможност да се създаде пълният скрипт за създаване на обектите. Предимствата на този подход са, че винаги е наличен скрипт, от който може да се създаде базата от данни. Възможно е и запазване на файловете, съставящи даден DB Project в система за контрол на изходния код, подобна на SourceSafe. По този начин се пази история за промените по базата от данни и има възможност за връщане на стара версия при неправилни промени.

Може би най-голямото удобство на проектите за бази от данни е възможността в тях да сложим съхранените процедури и да ги изпълняваме постъпково от средата на Visual Studio. По този начин можем да добавим към един сълюшън DB Project, съдържащ всички обекти на базата от данни, уеб услуга осъществяваща достъп до тази база от данни и клиентското приложение за тази услуга изградено на базата на Windows Forms или ASP.NET. Така всички части, необходими за изграждане на дадено приложение, ще вървят заедно.

Query Analyzer

Един от най-полезните инструменти, при създаване и настройка на SQL заявки за SQL Server е предлаганият от него инструмент - Query Analyzer.

Чрез него може лесно и бързо да се създават различни SQL заявки. Заявките могат да се пишат ръчно или да се създават на базата на шаблоните, които идват към продукта. Това става чрез привлачване на съответния шаблон от Templates страницата на Object Browser в работното пространство за въвеждане на SQL заявките.

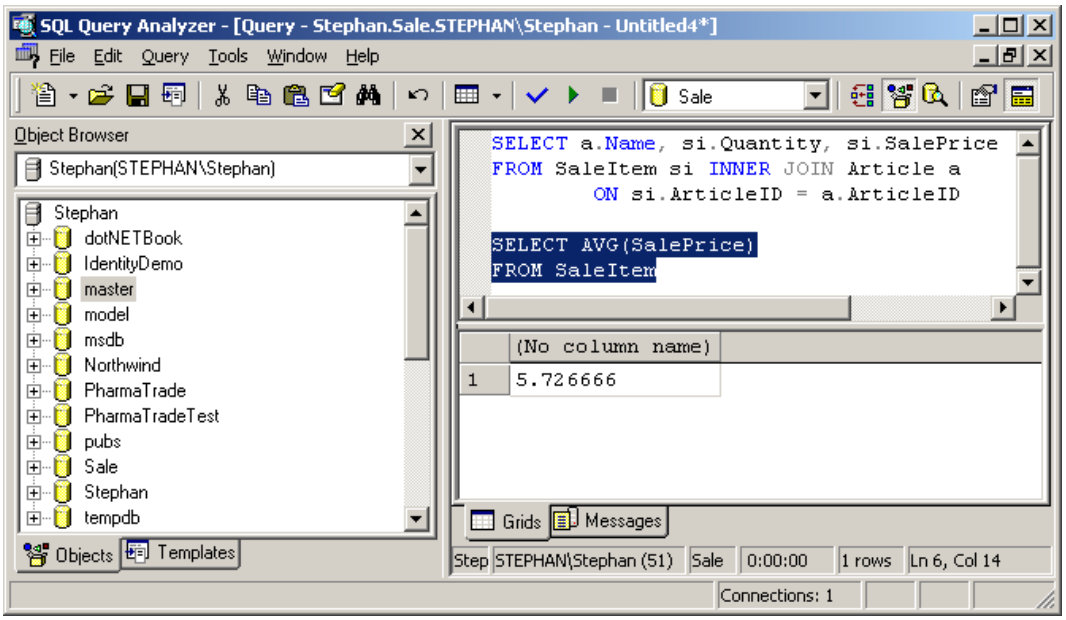
Друга полезна възможност, предлагана от Query Analyzer е дебъгването на съхранени процедури. Дебъгването на съхранени процедури не се различава от дебъгването на код на C# – дефинират се точки на прекъсване и се наблюдават определени стойности при постъпковото изпълнение. Стартирането на постъпковото изпълнение става чрез избиране на съхранената процедура от съответната база от данни и избор на Debug от контекстното меню, показващо се при натискане на десен бутон на мишката.

Не дебъгвайте съхранена процедура на сървър на който работят реални потребители. При този процес се заемат множество ресурси, което може да доведе до прекъсване работата на останалите потребители.

Query Analyzer има възможност и за фина настройка на производителността на дадена SQL заявка. Това става чрез разглеждане на т. нар. план на изпълнение (execution plan) и отстраняване на тесните места при

изпълнение на заявката от сървъра. Показването на плана на изпълнение става чрез предварително включване на опцията на Show Execution Plan, от менюто Query, която показва в графичен вид отделните стъпки при изпълнението на последно стартираната SQL заявка.

Момент от работата на Query Analyzer е показан на следващата картинка:



В прозореца за заявки са написани две заявки, които биха се изпълнили една след друга. Ако искаме да изпълним само една от тях трябва да я маркираме (както е показано на картинката) и да я стартираме чрез натискане на клавиша [F5] или избор на Execute от менюто Query.

Точно над прозореца за заявки се намира полето за избор, от което се определя върху коя база от данни ще се изпълнят въведените команди. В случая е избрана базата от данни **`Sale`**.

Понятието база от данни в SQL Server

Една база от данни в MS SQL Server се състои от множество таблици и други обекти като съхранени процедури, изгледи, тригери и т. н.

Всеки сървър може да съдържа няколко бази от данни, които са независими една от друга. Можем да имаме две таблици, с еднакви имена в две различни бази от данни.

Всяка база от данни има най-малко един файл, в който се съхраняват данните (data file) и един транзакционен файл (transaction log), в който се пази информация за всички направени по тях промени. Последният се използва при възстановяване на системата при срив.

На една машина могат да работят едновременно няколко SQL Server инстанции, всяка от които управлява няколко бази от данни.

Всяка инстанция на SQL Server съдържа една системна база от данни, която е наречена **master**. Тя съдържа специални таблици, които пазят системна информация, използвана от сървъра и е силно препоръчително да не се създават нови или модифицират съществуващите там таблици.

Въведение в T-SQL

Разширението на езика SQL, използвано в Microsoft SQL сървър, се нарича T-SQL (или още Transact-SQL). Този език основно предлага 3 типа команди: Data Definition Language (DDL), Data Manipulation Language (DML), DataBase Console Commands (DBCC), както и някой други възможности като системни съхранени процедури.

В настоящата тема няма да даваме подробни обяснения и примери за езика SQL и T-SQL, тъй като това би отнело обем с големината на една книга. Очаква се читателите да са запознати с SQL синтаксиса и с неговите възможности. Целта настоящото въведението в T-SQL е да предоставим кратък преглед на поддържаната от SQL Server функционалност, без да претендираме за изчерпателност.

Data Definition Language (DDL)

DDL включва команди за дефиниция и управление на обектите в базата от данни (създаване, промяна и изтриване на таблици, изгледи, тригери и т.н.). По-долу е илюстрирано използването на командите. За по-детайлно описание се обърнете към документацията на SQL Server и допълнителната литературата, посочена в края на темата.

CREATE

Командата **CREATE** се използва за създаване на обекти в базата данни, например нова база данни, таблица, изглед, съхранена процедура, тригер, индекс, ограничение и т.н.

Създаване на нова база данни

Да разгледаме първо как се създава нова база данни в SQL Server. Това става с командата **CREATE DATABASE**. По-долу е даден примерен SQL скрипт, който създава нова база данни с име **Sales**:

```
USE master
GO

CREATE DATABASE Sales
ON
( NAME = Sales_data,
  FILENAME = 'c:\mssql\data\Sales_data.mdf',
  SIZE = 10, MAXSIZE = 50, FILEGROWTH = 5 )
LOG ON
( NAME = 'Sales_log',
```

```
  FILENAME = 'c:\mssql\data\Sales_log.ldf',
  SIZE = 5MB, MAXSIZE = 25MB, FILEGROWTH = 5MB )
GO
```

Първият ред (командата **USE master**) от примера задава, че следващите команди ще се изпълняват в контекста на базата данни **master**, т.е. всички следващи команди като **INSERT**, **DELETE** и т.н. ще засягат обекти от посочената база от данни, освен ако не е посочена явно друга.

GO командата сигнализира на клиентските инструменти, че операторите преди нея до началото на скрипта или друга **GO** команда трябва да бъдат изпълнени в пакет. Изпълнението на командите в пакет намалява комуникацията между сървъра и клиента и повишава производителността.

Командата **CREATE DATABASE** от примера създава база данни с име **Sales**. Файлът, съхраняващ данните, се определя да бъде **Sales_data**, за който се задават начална големина 10MB, максимална големина 50 MB и стъпка на нарастване 5 MB. Файлът с дневника на транзакциите се задава да бъде **Sales_log** с начална големина 5 MB, максимална големина 25 MB и стъпка на нарастване 5 MB. Определянето на стъпките на нарастване на посочените файлове е важен момент за производителността на една база от данни, защото нарастването е бавен процес, който ако се случва често ще влоши производителността. Задаването на прекалено голяма стъпка на нарастване също е лош вариант, защото тогава времето за нарастване ще бъде прекалено голямо и ще спре за дълго време работата на сървъра.

Създаване на таблици

Нека сега видим как можем да създаваме таблици в дадена база данни в SQL Server и как можем да добавяме индекси към съществуващи таблици. Следващият пример създава таблица на име **Users** в базата от данни **Sales** и дефинира индекс за колоната, която съхранява първичния й ключ:

```
USE Sales
GO

CREATE TABLE Users
(
  UserID int identity NOT NULL,
  FirstName nvarchar(50),
  LastName nvarchar(50),
  Email varchar(50),
  Phone varchar(20),
  Phone2 varchar(20),
  Mobile varchar(20)
)
GO
```

```
CREATE CLUSTERED INDEX IDX_USERS_PK
ON Users (UserID ASC)
GO
```

Таблицата **Users** се състои от полетата **UserID**, **FirstName**, **LastName**, **Email**, **Phone**, **Phone2** и **Mobile**. Първото поле се задава, че ще съдържа целочислени стойности. Добавянето на **NOT NULL** към дефиницията на добавя ограничението сървъра да не допуска празни стойности за това поле, т.е. винаги трябва да има въведена стойност за него. Задаването на **identity** за дадена колона указва, че сървърът сам ще генерира последователни нарастващи стойности за съответното поле. Това означава, че първият запис в таблицата ще има **UserID = 1**, вторият **UserID = 2** и т.н. За съжаление обаче, не се гарантира че генерираната поредица от числа ще е непрекъсната, т.е. след 1 може да се зададе направо 3.

Останалите полета се задават, че ще съдържат символни данни, като в скобите е посочена максималната дължина на символния низ, който може да се съхранява. Полетата, зададени като **varchar** са характерни с това, че заемат физически толкова място в базата от данни, колкото е дължината на съхранявания от тях символен низ, а не колкото е била максималната обявена дължина при създаването на базата от данни.

За таблицата се създава индекс **IDX_USERS_PK** по полето **UserID**, който подрежда данните по нарастващ ред, заради добавената ключова дума **ASC**. Използването на **CLUSTERED** в дефиницията създава специален клъстерен индекс, който указва да се използва физическа подредба на записите в таблицата. При клъстерните индекси данните от таблицата се съхраняват вътре в самия индекс. За разлика от тях обикновените индекси са отделна физическа структура, поради което работят по-бавно.

ALTER

Командата **ALTER** се използва за промяна на обекти в базата от данни, например промяна на база от данни, промяна на таблица, промяна на съхранена процедура, промяна на ограничение, промяна на индекси т. н.

Чрез командата **ALTER**, например, е възможно да се промени дадена таблица, като се добави или изтрие дадена колона. Добавянето на колона става по начин, при който съществуващите до момента данни в останалите колони се запазват. Следващият пример модифицира създадената по-горе таблица **Users** като добавя колона с име **Pass** от тип **varchar** с дължина 100 символа.

```
USE Sales
GO

ALTER TABLE Users ADD Pass VARCHAR(100)
```

Ето и един пример за модификация на база данни. Той модифицира базата данни **Sales**, като добавя нов файл с име **Sales_idx.ndf**, в който ще се съхраняват данните постъпващи в тази база.

```
USE master
GO

ALTER DATABASE Sales
ADD FILE
(
  NAME = Sales_idx,
  FILENAME = 'd:\mssql\data\Sales_idx.ndf',
  SIZE = 5MB, MAXSIZE = 50MB, FILEGROWTH = 5MB
)
```

DROP

Командата **DROP** се използва за изтриване на обекти от базата от данни, например изтриване на база от данни, изтриване на таблица, премахване на индекс, изтриване на съхранена процедура, премахване на ограничение, изтриване на тригер и т.н.

Действието на командата ще бъде илюстрирано чрез два примера. Първият от тях изтрива таблицата **Users** от базата данни **Sales**, а вторият изтрива самата база данни **Sales**:

```
USE Sales
GO

DROP TABLE Users

USE master
GO
DROP DATABASE Sales
```

GRANT

Командата **GRANT** в SQL Server се използва за задаване на право на даден потребител (или роля) да извършва определени действия върху даден обект от базата данни. Например за един потребител може да се разреши добавянето на записи към дадена таблица, докато за друг това може да е забранено.

Следващият пример задава право на всеки потребител на базата данни да изпълнява командата **SELECT** върху таблицата **Users**:

```
USE Sales
GO

GRANT SELECT ON Users TO Public
```

```
GO
```

Ролята **Public**, която е използвана, обединява всички потребители в базата данни. Командата **GRANT** може да работи както с потребители, така и с роли.

DENY

Командата **DENY** в SQL Server се използва за отнемане на дадено право от потребител за достъп до даден ресурс. Примерът по-долу отнема правата на всички потребители за добавяне, изтриване и обновяване на таблицата **Users** в базата от данни **Sales**:

```
USE Sales
GO

DENY INSERT, UPDATE, DELETE ON Users TO Public
GO
```

REVOKE

Командата **REVOKE** се използва за отмяна на действието на командите **GRANT** или **DENY**. Използването на командата е илюстрирано чрез следващият пример:

```
USE Sales
GO

GRANT SELECT ON Users TO Public
DENY SELECT ON Users TO Guest
-- Guest account don't have permissions to execute SELECT
-- statement on Users table

REVOKE SELECT ON Users TO Guest
-- Guest account can execute again SELECT statements on
-- Users table
```

В примера **GRANT** командата задава права за изпълнение на **SELECT** заявки на всички потребители върху таблицата **Users**. След това чрез **DENY** се отнемат правата на потребителя **Guest** да изпълнява **SELECT** заявки върху таблицата **USERS**. Командата **REVOKE** връща правата на потребителя **Guest** за изпълнение на **SELECT** заявки.

Data Manipulation Language (DML)

DML включва команди за търсене, извличане, добавяне и изтриване на данни. Накратко в настоящата тема ще разгледаме действието на командите **SELECT**, **INSERT**, **UPDATE** и **DELETE**.

SELECT

Командата **SELECT** се използва за извличане на данни. Тя връща като резултат набор от записи от една или няколко таблици. Синтаксисът на командата е следният:

```
SELECT select_list
[INTO new_table_name]
FROM table_list
[WHERE search_conditions]
[GROUP BY group_by_list]
[HAVING search_conditions]
[ORDER BY order_list [ASC | DESC]]
```

SELECT е SQL командата, която е едновременно с най-сложен синтаксис и най-често използвана. Заградените в квадратни скоби клаузи са незадължителни. Отделните й елементи имат предназначение както следва:

- **select_list** – представлява списъка от колони, които ще се върнат като резултат от изпълнението на заявката. Може да се използва *, за да се окаже, че ще се върнат всички колони от засегнатите таблици.
- **new_table_name** – име на таблица, в която се записва резултатът от изпълнението на заявката. Тази таблица се създава автоматично и не трябва да съществува преди това в базата от данни. Схемата и се генерира автоматично по такъв начин, че да може да бъдат вмъкнати в нея върнатите като резултат данни.
- **table_list –** списък на таблиците, от които се извличат данните (колоните), връщани като резултат.
- **search_condition** (за **WHERE** клаузата) – определя условия, които трябва да удовлетворяват записите от изброените таблици за да се върнат като резултат от изпълнението на заявката.
- **group_by_list** – задава правила, по които записите удовлетворяващи условията, зададени в **search_condition**, да се групират. Веднъж групирани, върху тях може да се приложат различни агрегиращи функции. Агрегиращите функции ще бъдат обяснени по-късно в настоящата тема.
- **search_condition** (за **HAVING** клаузата) – задава условия за филтриране на групираните чрез **GROUP BY** записи.
- **order_list** – задава критерии, по които да бъдат подредени данните върнати от заявката. **ASC** и **DESC** задават съответно нарастващ и намаляващ ред. **ASC** се приема по подразбиране и може да се изпусне.

Примери за използването на SELECT

Без да навлизаме в детайли или да претендираме за изчерпателност, ще дадем няколко примера за използването на **SELECT** командата за извличане на данни от SQL Server база данни.

Пример за SELECT с ORDER BY клауза

Примерна **SELECT** заявка, която извлича колоните **FirstName**, **LastName** и **Mobile** от таблицата **Users**, като задава псевдонима **GSM** за последната колона, е дадена по-долу:

```
USE Sales
GO

SELECT FirstName, LastName, Mobile as GSM
FROM Users
ORDER BY LastName
```

Резултатът ще бъде сортиран по колоната **LastName** в нарастващ ред благодарение на **ORDER BY** клаузата.

Пример за SELECT с WHERE клауза

Пример за използването на **WHERE** клаузата е даден в следващата SQL заявка:

```
USE Sales
GO

SELECT FirstName, LastName, Mobile as GSM
FROM Users
WHERE LastName = 'Ivanov'
```

Резултатът от изпълнението на горната заявка ще бъде аналогичен на този в предходния пример, но ще бъдат върнати само записите, при които колоната **LastName** съдържа името **Ivanov**.

Пример за SELECT с GROUP BY клауза

Използването на клаузата **GROUP BY** ще бъде илюстрирано върху таблицата **SaleItem**, която има следната структура:

```
USE Sales
GO

CREATE TABLE SaleItem(
   SaleItemID int identity NOT NULL,
   ArticleID int NOT NULL,
   Quantity int NOT NULL,
```

```
   SalePrice decimal(9, 2) NOT NULL
)
GO

INSERT INTO SaleItem VALUES(1, 12, 5.40)
INSERT INTO SaleItem VALUES(3, 19, 6.48)
INSERT INTO SaleItem VALUES(1, 27, 5.30)
```

Като резултат от изпълнението на горния скрипт ще се получи следната таблица:

SaleItem

SaleItemID	ArticleID	Quantity	SalePrice
1	1	12	5.40
2	3	19	6.48
3	1	27	5.30

Тя ще съдържа данни за продажби на стоки. Полето **SaleItemID** е уникален идентификатор на продажбата (първичен ключ за таблицата). Той се генерира автоматично от SQL Server при добавяне на нов запис в таблицата заради наличието на **identity** при декларацията на полето. **ArticleID** и **Quantity** съдържат съответно кода на стоката и продаденото количество. Полето **SalePrice** съдържа продажната цена с точност до втория знак след десетичната точка.

Командата **INSERT INTO** в горния пример има за задача да добави записи към новосъздадената таблица **SaleItem**. Нейното действие ще бъде разгледано малко по-късно в настоящата тема.

Чрез досега разгледаните възможности на **SELECT** заявката може да извлечем данни за всички продажби на дадена стока. Ако обаче искаме да изпълним заявка, която обединява данни за продажбите на няколко стоки в една заявка трябва да използваме **GROUP BY**.

Следващият пример намира общия брой на продаденото количество за всяка стока.

```
USE Sales
GO

SELECT ArticleID, SUM(Quantity) TotalSales
FROM SaleItem
GROUP BY ArticleID
```

Резултатът от изпълнението на заявката ще съдържа две колони – **ArticleID** и **TotalSales**, съответно за кода на стоката и общия брой на продажбите й. Намирането на общия брой на продажбите се извършва чрез функцията **SUM**, която сумира стойностите на колоната, подадена й

като аргумент (в случая **Quantity**). Тъй като чрез **GROUP BY** клаузата е указано да се извърши групиране по колоната **ArticleID**, то сумирането на продадените количества ще се извърши за всяка стока поотделно. Като резултат ще се върнат следните данни:

ArticleID	TotalSales
1	39
3	19

Пример за SELECT с GROUP BY и HAVING клауза

Ако искаме да намерим всички стоки, чиято бройка на продажбите е над определен брой (например 25), ще трябва да използваме **HAVING** клаузата на **SELECT** заявката. В нея подобно на **WHERE** се указват ограничения, които обаче се отнасят за групирането чрез **GROUP BY**. Следващата примерна заявка връща само реда с код на стоката 1, защото продажбите за нея са над определения брой – 25.

```
USE Sales
GO

SELECT ArticleID, SUM(Quantity) TotalSales
FROM SaleItem
GROUP BY ArticleID
HAVING SUM(Quantity) > 25
```

Агрегиращи (обобщаващи) функции

В последния пример използвахме една типична агрегираща SQL функция: **SUM(…)**. Агрегиращите функции връщат стойност в зависимост от съдържанието на даден стълб. Някой от най-важните такива функции са:

- **COUNT(…)** – връща броя на редовете, които удовлетворяват условието, наложено в **WHERE** клаузата.
- **SUM(…)** – връща сумата от стойностите в дадена колона.
- **AVG(…)** – връща средноаритметичната стойност от стойностите на дадена колона.
- **MAX(…)** – връща максималната стойност, съдържаща се в дадена колона.
- **MIN(…)** – връща минималната стойност, съдържаща се в дадена колона.

Агрегиращи (обобщаващи) функции – примери

Ще дадем няколко примера за използване на агрегиращи функции. Намирането на общия брой на продажбите (да не се бърка с коли-

чествата) за дадена стока с код 1 може да се извърши чрез следната заявка:

```
USE Sales
GO

SELECT Count(ArticleID)
FROM SaleItem
WHERE ArticleID = 1
```

Следващата заявка намира средноаритметичната цена на всички продажби за всички стоки:

```
USE Sales
GO

SELECT AVG(SalePrice)
FROM SaleItem
```

Намирането на максималната продажна цена за стока с код 1 става чрез следващата заявка:

```
USE Sales
GO

SELECT MAX(SalePrice)
FROM SaleItem
WHERE ArticleID = 1
```

Съединение (JOIN) на таблици

Под термина "**съединения на таблици**" се има предвид комбиниране на колони на две или повече таблици и връщането им във виртуална таблица, при изпълнение на командата **SELECT**. Съединението на таблиците може да се раздели на следните категории: вътрешно съединение (INNER JOIN), външно съединение (OUTER JOIN) и кръстосано съединение (CROSS JOIN).

Различните видове съединения ще бъдат демонстрирани с помощта на таблиците **SaleItem**, която беше показана вече при описанието на **SELECT** и таблицата **Article** която има следната структура:

```
USE Sales
GO

CREATE TABLE Article(
   ArticleID int identity NOT NULL,
   Name varchar(40) NOT NULL
)
```

```
INSERT Article VALUES('Zagorka');
INSERT Article VALUES('Tuborg');
INSERT Article VALUES('Amstel');
```

Като резултат от изпълнението на горния скрипт ще се получи следната таблица:

Article

ArticleID	Name
1	Zagorka
2	Tuborg
3	Amstel

Вътрешно съединение (INNER JOIN)

Вътрешното съединение между две таблици връща редовете от първата таблица, само ако те имат съответствие с редовете от втората таблица, участваща в съединението.

Следващата заявка използва вътрешно съединение за да извлече имената на стоките, количеството и продажната цена от таблиците **Article** и **SaleItem**:

```
USE Sales
GO

SELECT a.Name, si.Quantity, si.SalePrice
FROM SaleItem si INNER JOIN Article a
     ON si.ArticleID = a.ArticleID
```

Връзката, чрез която се свързват двете таблици, са полетата **ArticleID** от всяка една от тях. **ArticleID** в таблицата **SaleItem** е външен ключ към таблицата **Article**, а **ArticleID** в таблицата **Article** е нейният първичен ключ.

Аналогичен на резултата, върнат от горната заявка, може да се получи и без да се използва ключовата дума **INNER JOIN**, като се приложи ограничение чрез **WHERE** клауза:

```
USE Sales
GO

SELECT a.Name, si.Quantity, si.SalePrice
FROM SaleItem si, Article a
WHERE si.ArticleID = a.ArticleID
```

Резултатът от изпълнението на горния скрипт ще бъде:

Name	Quantity	SalePrice
Zagorka	12	5.40
Zagorka	27	5.30
Amstel	19	6.48

Както се вижда, има няколко начин за извличане на данни от таблици, свързани една с друга посредством връзка 1 към много. Ако се чудите кога да използвате съединения и кога да ги имитирате чрез **WHERE** клаузата, препоръката е да предпочитате съединенията.

Външни съединения (OUTER JOIN)

Външните съединения биват няколко типа: ляво външно съединение, дясно външно съединение и пълно външно съединение. Нека ги разгледаме поотделно.

Ляво външно съединение (LEFT OUTER JOIN)

Лявото външно съединение между две таблици връща всички редове от първата таблица плюс съответстващите им редове от втората таблица. Ако във втората таблица няма съответстващи редове, то колоните от втората таблица се връщат със стойност **NULL**.

Нека да разгледаме по-детайлно разликата между вътрешното и лявото външно съединение на таблици. За целта ще направим left outer join между **SaleItem** и **Article** и ще сравним резултата с този при използване на INNER JOIN.

```
use Sales
GO

SELECT a.Name, si.Quantity, si.SalePrice
FROM Article a LEFT OUTER JOIN SaleItem si
     ON a.ArticleID = si.ArticleID
```

Резултатът от изпълнението на горната заявка ще бъде:

Name	Quantity	SalePrice
Zagorka	12	5.40
Zagorka	27	5.30
Tuborg	NULL	NULL
Amstel	19	6.48

В горната таблица забелязваме, че лявото външно съединение на таблиците **Article** и **SaleItem** връща един ред повече от вътрешното съединение, показано в предишната точка на настоящата тема. Допълнителният ред е за стоката с име **Tuborg**, за която **Quantity** и **SalePrice** са

NULL. Това се дължи на факта, че във втората таблица – **SaleItem** няма съответстващи записи за стока с код 2. Полетата, идващи от нея, при съединението се връщат със стойност **NULL**. При вътрешното съединение, редовете за които няма съответствие в двете таблици се пропускат. Затова при двете съединения се получава разлика от един ред.

Дясно външно съединение (RIGHT OUTER JOIN)

Дясното външно съединение между две таблици връща всички редове от втората таблица плюс съответстващите им редове от първата таблица. Ако в първата таблица няма съответстващи редове, то колоните от първата таблица се връщат със стойност **NULL**.

Резултатът, получен от изпълнението на лявото външно съединение на таблиците **SaleItem** и **Article**, може да се получи и чрез дясно външно съединение, чрез следната заявка:

```
USE Sales
GO

SELECT a.Name, si.Quantity, si.SalePrice
FROM SaleItem si RIGHT OUTER JOIN Article a
     ON si.ArticleID = a.ArticleID
```

Пълно външно съединение (FULL OUTER JOIN)

Пълното външно съединение между две таблици връща всички редове от двете таблици, за които има съответствие. Към тях се прибавят редовете от първата таблица за които няма съответствие във втората таблица, като редовете от втората таблица се връщат със стойност **NULL**. Към тях се прибавят и редовете от втората таблица, за които няма съответствие в първата таблица. Редовете от първата таблица се връщат със стойност **NULL**.

Пълното външно съединение представлява нещо като комбинация от ляво външно и дясно външно съединение. Ето един пример:

```
USE Sales
GO

SELECT a.Name, si.Quantity, si.SalePrice
FROM SaleItem si FULL OUTER JOIN Article a
     ON si.ArticleID = a.ArticleID
```

Кръстосано съединение (CROSS JOIN)

Кръстосаното съединение между две таблици връща комбинация на редовете от таблиците, участващи в съединението (декартово произведение).

Нека си представим, че имаме две таблици, първата от които съхранява първо име, а втората фамилия на хора. Ако искаме да получим всички

възможни комбинации от пълните имена, които може да се генерират чрез данните, съхранени в двете таблици, ще използваме кръстосано съединение. Ето илюстрация на описания пример:

```
USE Sales
GO

CREATE TABLE FirstName(
   Name varchar(40) NOT NULL
)

CREATE TABLE LastName(
   Name varchar(40) NOT NULL
)
GO

INSERT INTO FirstName VALUES ('Stephan')
INSERT INTO FirstName VALUES ('Stoimen')

INSERT INTO LastName VALUES ('Zahariev');
INSERT INTO LastName VALUES ('Ivanov');
GO

SELECT fn.Name FirstName, ln.Name LastName
FROM FirstName fn CROSS JOIN LastName ln
```

В показания SQL скрипт първо се създават таблиците **FirstName** и **LastName**, всяка от които съдържа по една колона от тип символен низ. След това във всяка от таблиците се вмъкват по два записа. Изпълнението на **SELECT** заявката, която използва вътрешно съединение за да свърже двете таблици, дава като резултат следните данни:

FirstName	LastName
Stephan	Zahariev
Stoimen	Zahariev
Stephan	Ivanov
Stoimen	Ivanov

Комбинация от съединения между таблици

В една **SELECT** заявка може да се прилагат няколко различни съединения между таблици, както и да се съединяват повече от две таблици едновременно. Пример за такава заявка, ще дадем използвайки показаните подолу таблици **Author**, **Book** и **Country** съдържащи данни за автори, написаните от тях книги и националността на авторите:

Country

CountryID	Name
1	Bulgaria
2	Italy
3	USA

Author

AuthorID	Name	CountryID
1	Marco Cantu	2
2	Jeffrey Richter	3
3	Ivan Ivanov	1

Book

BookID	Name	AuthorID
1	Mastering Delphi	1
2	Applied Microsoft .NET Framework Programming	3

Ако искаме да извлечем данните от трите таблици, така че да покажем всички автори, написаните от тях книги и националността им трябва да използваме показаната по-долу примерна заявка използваща съединение на трите таблици:

```
SELECT a.Name Author, c.Name Country, b.Name Book
FROM Author a
   INNER JOIN Country c ON c.CountryID = a.CountryID
   LEFT OUTER JOIN Book b ON b.AuthorID = a.AuthorID
```

Към таблицата **Author** чрез вътрешно съединение свързваме таблицата **Country**, защото всеки автор в примерната база от данни има една националност. Използваме ляво външно съединение за да присъединим и таблицата **Book**, защото за даден автор в нея може да няма все още запис. Резултатът от изпълнението на заявката ще бъде:

Author	Country	Book
Marco Cantu	Italy	Mastering Delphi
Jeffrey Richter	USA	Applied Microsoft .NET Framework Programming
Ivan Ivanov	Bulgaria	NULL

INSERT

Командата **INSERT** добавя нов ред към дадена таблица. Нейният синтаксис е следният:

```
INSERT [INTO] TableName[(ColumnList)] VALUES (ValuesList)
```

INTO представлява ключова дума, която е незадължителна и може да се пропусне. **TableName** е името на таблицата в която ще добавяме нов запис, а **ColumnList** е списък от нейните колони. Този списък може да не включва всички колони на таблицата. Тогава при добавянето, пропусна-

тата колона ще се добави със стойност **NULL** или ще получи стойността по подразбиране, обявена при създаване на таблицата. Ако е пропуснат целият списък, се подразбира, че ще се добавя стойност във всички колони на таблицата. **ValuesList** съдържа списъка от стойности, които ще се добавят. Този списък трябва да отговаря по брой на колоните с този на **ColumnList** или на всички колони от таблицата ако **ColumnList** е пропуснат.

Следващият SQL скрипт добавя един запис към таблицата **Users**:

```
USE Sales
GO

INSERT Users
   (FirstName, LastName, Phone, Mobile, Email)
VALUES
   ('Ivan', 'Ivanov', '997567', '+359 88 123 4567',
   'Ivan@abv.nospam.bg')
```

UPDATE

Командата **UPDATE** се използва за обновяване на даден запис (или група от записи) в дадена таблица. Синтаксисът й е следният:

```
UPDATE TableName
SET
   colname1=value1,
   colname2=value2
[WHERE Condition]
```

TableName представлява името на таблицата, в която ще обновяваме записи. **colname1**, **colname2** и т. н. са имената на колоните в таблицата, които ще обновяваме. **value1**, **value2** са новите стойности съответно на **colname1** и **colname2**. Една **UPDATE** команда може да модифицира както само една колона на дадена таблица, така и всички нейни колони. **Condition** посочва критериите, на които трябва да отговарят записите за да бъдат модифицирани. Ако налагането на това условие се изпусне, то тогава се обновяват всички записи в посочената таблица.

Следващият пример обновява таблицата **Users** като променя e-mail адреса на потребител с идентификатор 118.

```
USE Sales
GO

UPDATE Users
SET Email='Ivan@dir.nospam.bg'
WHERE UserID = 118
```

Обновяването на дадена таблица се счита за атомарна операция, т.е. промените изискани от командата **UPDATE** или изцяло се извършват или се отказват всички. Нека да имаме заявка, която обновява 10 000 реда, като поради някаква грешка (например нарушаване на ограничение) последния ред не може да се обнови. Тогава промените направени по останалите 9999 реда се отказват и командата връща грешка.

Обикновено след като се изпълни команда **UPDATE** се проверява състоянието кода, връщан от системните функции **@@ERROR** и **@@ROWCOUNT**, които съдържат кода за грешка на последно изпълнената SQL команда и броя на засегнатите от нея редове.

DELETE

Командата **DELETE** се използва за изтриване на записи от дадена таблица. Нейният синтаксис е следният:

```
DELETE [FROM] TableName
[WHERE Condition]
```

Ключовата дума **FROM** е незадължителна и може да се изпусне. **TableName** посочва името на таблицата, от която ще се изтриват записи. **Condition** съдържа критериите, на които трябва да отговарят записите за да бъдат изтрити.

Ето един пример, който илюстрира използването на командата **DELETE**:

```
USE Sales
GO

DELETE FROM Users
WHERE UPPER(LastName) = 'IVANOV' AND Mobile LIKE '+359%'
```

Примерът изтрива от таблицата **Users** всички записи, за които колоната **LastName** има стойност **Ivanov** и колоната **Mobile** започва с **+359**.

Подобно на командата **UPDATE**, **DELETE** също се счита за атомарна операция и се изтриват всички или нито един от посочените записи.

Когато пишете UPDATE или DELETE команди, винаги проверявайте дали сте наложили ограничение чрез WHERE клаузата. В противен случай командата ще се изпълни върху цялата таблица, което рядко е желаният ефект.

DBCC команди в SQL Server

SQL Server предлага така наречените Database Console Commands (DBCC). Те се използват за проверка на физическата и логическата консистентност на базата от данни. Могат да се обособят в следните групи:

DBCC команди за поддръжка

- **DBCC DBREINDEX –** изгражда отново един или няколко индекса на таблица.
- **DBCC INDEXDEFRAG –** дефрагментира клъстерни или вторични индекси за зададената таблица или изглед.
- **DBCC SHRINKDATABASE –** намалява размера на файловете с данни за дадена база от данни.
- **DBCC SHRINKFILE –** намалява размера на зададения файл с данни или log файл за дадена база от данни.
- **DBCC UPDATEUSAGE –** докладва и отстранява неточности в таблицата **sysindexes**. Таблицата **sysindexes** е системна за SQL Server и съдържа по един запис за всеки индекс и таблица в базата от данни.

DBCC команди за проверка на статуси

- **DBCC OPENTRAN –** показва информация за най-старата активна (незавършена) транзакция.
- **DBCC INPUTBUFFER –** показва последния SQL израз изпратен от към SQL Server за изпълнение.
- **DBCC OUTPUTBUFFER –** показва последните резултати от изпълнението на даден SQL израз изпратени към клиента заявил изпълнението.
- **DBCC PROCCACHE –** извежда информация в табличен вид за процедурния кеш. Процедурния кеш се нарича област от паметта, където се пази плана за изпълнение на заявките.
- **DBCC SQLPERF –** показва статистика за използваното дисково пространство от Transaction Log файловете на всички бази от данни.
- **DBCC USEROPTIONS –** връща активните SET опции за текущата връзка.

DBCC команди за валидация целостта на данните

- **DBCC CHECKALLOC –** проверява интегритета на структурите за заделяне на дисково пространство.
- **DBCC CHECKCATALOG –** проверява интегритета в и между системните таблици в дадена база от данни.
- **DBCC CHECKCONSTRAINTS –** проверява интегритета на зададените ограничения на дадена таблица в базата от данни.
- **DBCC CHECKDB –** проверява разпределението и структурния интегритет на всички обекти в посочената база от данни.
- **DBCC CHECKFILEGROUP –** проверява разпределението и структурния интегритет на всички таблици в посочената файлова група.

- **DBCC CHECKIDENT –** проверява всички колони, които са декларирани като **identity** и ако е необходимо извърша корекция.
- **DBCC CHECKTABLE –** проверява интегритета на данните, индексите и т. н. за дадена таблица.

Други DBCC команди

- **DBCC DllName (FREE) –** премахва от паметта зададен **.dll** файл, който съдържа разширена съхранена процедура. Разширените съхранени процедури се пишат на език като C++.
- **DBCC HELP –** връща информация за синтаксиса на даден DBCC израз.
- **DBCC PINTABLE –** указва на SQL Server да не премахва страниците на дадена таблица от паметта. По този начин обработката на таблицата ще става в паметта а не на твърдия диск.
- **DBCC TRACEON / TRACEOFF –** разрешава/забранява зададения флаг за трасиране.
- **DBCC UNPINABLE –** премахва указанието наложено от **DBCC PINTABLE** за поддържане страниците на дадена таблица в паметта. При нужда тези страници може да бъдат премахнати от паметта.

Представеното по-горе обяснение на DBCC командите е съвсем бегло. За по-детайлно тяхно описание се обърнете към цитираната литература в края на главата.

Примери за използване на DBCC команди

Показаният по-долу пример използва командата **DBCC SQLPERF** за да покаже статистиката за използваното дисково пространство от Transaction Log файловете на всички бази от данни.

```
USE Master
GO

DBCC SQLPERF (LOGSPACE)
```

Следващият пример проверява базата от данни **Master** за грешки в интегритета:

```
USE Sales
GO

DBCC CHECKDB
```

Съхранени процедури

Както вече изяснихме, съхранените процедури представляват програмен код, състоящ се от последователност от SQL команди, които се изпълняват

в самия сървър за бази от данни. SQL Server поддържа следните типове съхранени процедури:

- **системни** – това са съхранени процедури, които се използват за административни цели. Обикновено се извикват от инструменти като Enterprise Manager, но може да се използват и директно. Системните съхранени процедури започват с префикса **sp_**.
- **разширени** – представляват подпрограми, съдържащи се в динамични библиотеки (**.dll** файлове) и написани на езици като C и C++, които се зареждат и изпълняват като обикновени съхранени процедури. Обикновено започват с префикса **xp_**.
- **потребителски** – това са съхранени процедури, които са създадени от разработчиците или администраторите на един сървър. те се използват много често в практиката и затова трябва да им обърнем повече внимание.

Силно се препоръчва да не използвате префикса sp_ в името на създаваните от вас съхранени процедури. Ако SQL Server срещне такъв префикс в името на процедура, при извикването и той започва да претърсва за нея в някой специални бази от данни.

Създаването на съхранени процедури в SQL Server се извършва чрез командата **CREATE PROC** (**CREATE PROCEDURE**). Нейният синтаксис, който в случая е опростен с цел по-лесно възприемане, е следният:

```
CREATE PROC[EDURE] procedure_name
   [ { @parameter data_type }
     [VARYING] [=default] [OUTPUT] ]
   [ ,...n ]
AS
   sql_statement [ ...n ]
```

Заградените в квадратни скоби елементи са незадължителни, като отделните елементи имат следното значение:

- **procedure_name** – име на съхранената процедура.
- **@parameter** – параметър, които приема процедурата. Името на параметъра трябва винаги да започва със символа **@**.
- **data_type** – тип на параметъра.
- **VARYING** – тази ключова дума се слага към параметри от тип курсор. Курсорите са извън обхвата на настоящата тема. Повече информация за тях можете да намерите в препоръчаната литература в края на главата.
- **default** – задава подразбираща се стойност за параметъра, ако процедурата бъде извикана без параметри.

- **OUTPUT** – указва, че маркираният с тази ключова дума параметър е изходен, т.е. чрез него съхранената процедурата връща данни.
- **sql_statement** – един или няколко T-SQL израза, които реализират поведението на съхранена процедура.

Една съхранена процедура може да се обръща към други съхранени процедури или да извиква сама себе си рекурсивно. Това обаче може да се прави докато не се превиши лимита от 32 нива на влагане. Нивото на влагане се следи от специалната функция **@@NESTLEVEL**, която съдържа текущата дълбочина на влагане във всеки един момент.

Командата EXEC

Извикването на съхранени процедури се извършва чрез командата **EXEC** (**EXECUTE**), която има следния синтаксис, който отново е съкратен с цел опростяване:

```
[ [ EXEC [ UTE ] ]
   {
      [ @return_status = ]
        procedure_name
   }
   [ [ @parameter = ] { value | @variable [ OUTPUT ] | [ DEFAULT
] ]
      [ ,...n ]
```

Отделните елементи имат следното значение:

- **@return_status** – целочислена променлива, която ще получи кода, съдържащ статуса от изпълнението на съхранената процедура.
- **procedure_name** – име на съхранената процедура, която ще се извика.
- **@parameter** – име на параметър на съхранената процедура, дефиниран при създаването и.
- **value** – стойност на параметъра.
- **@variable** – име на променлива, която съдържа параметъра подаван на процедурата или получава стойност от изходен параметър.
- **OUTPUT** – ключова дума, маркираща че параметъра е изходен.
- **DEFAULT** – ключова дума, която указва да се вземе стойността по подразбиране за параметъра зададена при създаването на процедурата.

Синтаксисът на командата **EXEC** не изисква изписването на ключовата дума **EXEC** или **EXECUTE**, когато извикването на съхранената процедура се намира на първо място в подадения за изпълнение пакет на SQL Server.

Примери за съхранени процедури

За да илюстрираме командите, които разгледахме, ще дадем няколко примера за съхранени процедури в SQL Server.

Съхранена процедура, връщаща набор от данни

Нека да вземем примерната SQL заявка, показваща използването на командата **SELECT** заедно с клаузите **HAVING** и **GROUP BY**, и да видим как тя може да се преработи така, че данните да се връщат от съхранена процедура. Резултатът е показан по-долу:

```
USE Sales
GO

CREATE PROC spGetArticleSales (@MinQuantity int)
AS
   SELECT ArticleID, SUM(Quantity) TotalSales
   FROM SaleItem
   GROUP BY ArticleID
   HAVING SUM(Quantity) > @MinQuantity
GO
```

Съхранената процедура **spGetArticleSales** изпълнява просто една **SELECT** заявка, която връща всички стоки, от които са направени определено количество продажби. Въпросното количество продажби се задава чрез параметъра **@MinQuantity**. Извикването на съхранената процедура, така че да върне всички стоки с минимално продадено количество 25 става по следният начин:

```
USE Sales
GO

spGetArticleSales @MinQuantity=25
GO
```

Съхранена процедура, добавяща нов ред към таблица

Следващата примерна съхранена процедура добавя нов запис към таблицата **Article** и връща неговия автоматично генериран идентификатор.

```
USE Sales
GO

CREATE PROC spInsertArticle (@Name varchar(40), @ArticleID int
OUT)
AS
   SELECT @ArticleID = -1;

   INSERT INTO Article (Name) VALUES (@Name);
```

```
   IF(@@ERROR <> 0)
     RETURN @@ERROR;

   SELECT @ArticleID = @@IDENTITY;

   RETURN @@ERROR;
GO
```

Името на стоката, която ще се вмъкне в таблицата **Article** се получава в параметъра **@Name**. Параметърът **@ArticleID** е изходен и чрез него се връща идентификатора на добавения от процедурата запис.

В началото, съхранената процедура инициализира връщаната стойност за **@ArticleID** на -1, след което чрез командата **INSERT** добавя нов ред към таблицата **Article**. Тъй като колоната **ArticleID** от тази таблица е маркирана като **identity**, тя се пропуска при изброяване на колоните в **INSERT** командата. Нека да припомним, че **identity** колоните получаваха автоматично стойности, задавани от SQL Server.

След командата за добавянето на новия запис, се извърша проверка за това дали самото добавяне е било успешно. Ако то се е провалило поради някаква причина, съхранената процедура записва в изходния параметър **@ArticleID** стойност -1 и връща като резултат кода на грешката, който се намира в **@@ERROR**, след което прекратява работата си.

Ако добавянето е било успешно, изходният параметър **@ArticleID** получава стойността на колоната **ArticleID** за току-що добавения запис. Тази стойност се получава от системната функция **@@IDENTITY**. След това съхранената процедура връща стойността намираща се в **@@ERROR**, която при липса на грешка е 0 и прекратява работата си.

Извикването на описаната по-горе съхранената процедура може да се извърши по следния начин:

```
DECLARE @ArticleID int;

EXEC spInsertArticle 'Aspirin', @ArticleID OUTPUT
```

Първият ред от показаният по-горе код декларира променлива от тип **int** с име **@ArticleID**. Вторият ред извиква съхранената процедура **spInsertArticle**, която ще добави нова стока с име "Aspirin" и ще върне уникалния идентификатор на новодобавения ред в **@ArticleID**.

Транзакции в SQL Server

Идеологията на транзакциите и свързаните с тях термини като заключвания и нива на изолация бяха дискутирани вече в настоящата тема. Сега ще разгледаме специфичната им поддръжка от страна на SQL Server. Някой от предлаганите от него възможности, като именувани транзакции

и точки на записване на транзакциите са извън обхвата на темата, но читателят може да ги проучи в предложената допълнителна литература.

SQL Server поддържа два вида транзакции – **локални** и **разпределени**.

Разпределени транзакции

Разпределените транзакции се разпростират върху няколко сървъра – при тях или извършваните операции над базите от данни на всички сървъри се изпълняват успешно, или всички операции върху всеки от сървърите се отказват.

Разпределените транзакции се използват при големи и сложни разпределени системи, които работят върху няколко сървъра. В Windows и SQL Server разпределените транзакции се управляват от т. нар. Distributed Transaction Coordinator (DTC).

В настоящата тема няма да разглеждаме разпределените транзакции в детайли. Вместо това ще се фокусираме върху локалните транзакции, които са и най-често използваните.

Локални транзакции

Стартирането на нова транзакция в SQL Server се извършва чрез командата **`BEGIN TRANSACTION`** (или съкратения вариант **`BEGIN TRAN`**). Следващите я команди (като **`INSERT`** и **`UPDATE`**) се смятат за част от транзакцията и направените от тях промени се записват в базата от данни само при нейното потвърждаване. Това потвърждаване се извършва от командата **`COMMIT TRANSACTION`** (или съкратено **`COMMIT TRAN`**). Отмяна на действията извършени в транзакцията се извърша от командата **`ROLLBACK TRANSACTION`** (съкратения вариант е **`ROLLBACK TRAN`**). За да изпълните командите за отказ или потвърждение на транзакция, такава трябва да бъде стартирана. В противен случай SQL Server връща съобщение за грешка.

Вложени транзакции

Транзакциите могат да се влагат една в друга, т.е. изпълнението на една транзакция започва преди да е завършила предхождащата я. Ситуацията е сходна с извикването между методи на езици като C#. Концепцията за обработката на вложени транзакции, може да бъде на пръв поглед малко странна и затова бъдете сигурни, че разбирате добре следващите редове.

Влагането на транзакции се извършва просто чрез изпълняване на две команди **`BEGIN TRAN`**, между които не се среща **`ROOLBACK TRAN`** или **`COMMIT TRAN`**. Нивото на влагане на транзакциите се следи чрез системната функция **`@@TRANCOUNT`**, която връща броя на активните транзакции за текущата връзка. При изпълнение на две команди за стартиране на транзакция една след друга, **`@@TRANCOUNT`** ще върне 2.

При срещане на команда **`COMMIT TRAN`**, когато е стартирана вложена транзакция, SQL Server потвърждава вътрешната транзакция, като за

потвърждаване на външната ще е необходима още една команда **COMMIT TRAN**.

При срещане на команда **ROOLBACK TRAN**, когато е стартирана вложена транзакция, SQL Server отказва както вътрешната, така и външната транзакция. Тоест, с една команда отказваме промените направени от двете транзакции и в системата вече няма активна транзакция. Последващото изпълнение на командите **ROOLBACK TRAN** или **COMMIT TRAN**, без да бъде стартирана нова транзакция, ще доведе до грешка.

Транзакции и съхранени процедури

При работа с транзакции в съхранени процедури, SQL Server проверява връщаната стойност от системната функция **@@TRANCOUNT** и следи тя да бъде една и съща в началото и края на изпълнението на съхранената процедура. Ако двете стойности не съвпадат, извикването на процедурата връща грешка.

Изолация на транзакциите

Изолацията на транзакциите се задава чрез командата **SET TRANSACTION ISOLATION LEVEL**, която има следния синтаксис:

```
SET TRANSACTION ISOLATION LEVEL
  { READ COMMITTED
    | READ UNCOMMITTED
    | REPEATABLE READ
    | SERIALIZABLE
  }
```

Вече разгледахме изброените по-горе нива на изолация. Задаването на едно от тези нива на изолация се взема предвид от следващата команда за стартиране на транзакция и не влияе на нивото на изолация на текущо изпълняваната транзакция.

Заключвания и транзакции

По време на работа си SQL Server, заключва дадени записи в базата от данни с цел да гарантира изолацията на транзакциите. Заключванията, които се налагат върху записите могат да се разделят най-просто на два вида – **поделени** и **монополни**.

При поделените заключвания, заключеният запис може да се чете от всички транзакции, но не може да се редактира от тях.

При монополното заключване, записът може да се чете и редактира само от транзакцията, наложила заключването.

Видът заключвания, които SQL Server прави зависи от изолацията, която е зададена за изпълняваните транзакции. По-високите нива на изолация предизвикват по-сериозни заключвания.

Естествено, както във всяка система, използваща заключване, възниква проблемът с т. нар. "мъртва хватка" (dead lock). Типичен пример е ситуацията, при която транзакция 1 е заключила таблица A за четене и се опитва да модифицира данните в таблица B, а в същото време транзакция 2 е заключила таблица B и се опитва да модифицира данните в таблица A.

За щастие SQL Server автоматично открива възникването на ситуация от горния тип и автоматично отказва едната транзакция. Въпреки това, попадането в това положение е нежелателно и за да се избегне, достъпът до таблиците трябва да става в един и същи ред.

Възможно е възникването и на т. нар. "конверсни" заключвания, при които две транзакции са заключили дадена ред от таблицата за четене и след това едновременно се опитват да го редактират. Тази ситуация се избягва като се сложи подсказка за заключване (**UPDLOCK**) на **SELECT** командата, но това също е извън обхвата на настоящата тема.

Примери за употреба на транзакции

Нека разгледаме няколко примера за използването на транзакции в SQL Server и да демонстрираме възможностите на езика T-SQL за работа в транзакция.

Пример за транзакция в съхранена процедура

Цялостен пример за употреба на транзакции в комбинация с използване на съхранена процедура беше даден и описан подробно в предишната точка, разглеждаща релационните бази от данни. Затова сега ще дадем няколко други примера.

Пример за вложени транзакции

Следващият пример добавя две нови стоки към таблицата **Article**, като първата се добавя в контекста на външната транзакция, докато втората стока се добавя в контекста на вложената транзакция.

```
USE Sales
GO

BEGIN TRANSACTION

INSERT INTO Article(Name)
VALUES ('Zagorka')

IF @@ERROR = 0
BEGIN
   BEGIN TRANSACTION --Start inner transaction

   INSERT INTO Article(Name)
   VALUES ('Tuborg')
   IF @@ERROR = 0
```

```
    COMMIT TRANSACTION --commit inner transaction
  ELSE
    ROLLBACK TRANSACTION --rollback both transactions

  --commit outer transaction, if previous
  --insert was successfull
  IF @@TRANCOUNT = 1
    COMMIT TRANSACTION
END
ELSE
  ROLLBACK TRANSACTION --rollback outer transaction
```

Пример за промяна изолацията на транзакция

Показаният по-долу пример стартира транзакция с ниво на изолация **`REPEATABLE READ`** и добавя нов запис към таблицата **`Article`**. При успешно изпълнение на командата **`INSERT`**, транзакцията се потвърждава, а в противен случай се отказва.

```
USE Sales
GO

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

BEGIN TRANSACTION

INSERT INTO Article(Name)
VALUES ('Загорка')
IF @@ERROR = 0
  COMMIT TRANSACTION
ELSE
  ROLLBACK TRANSACTION
```

Пример, демонстриращ ефекта от промяна нивото на изолация на транзакция

Промяната на нивото на изолация на работа на дадена транзакция може да има драстични последици върху нейната работа. Нека да се убедим в това, чрез един нагледен пример. За целта стартираме Query Analyzer и се свързваме към SQL Server. Изпълняваме показания по-долу SQL, код за да стартираме транзакция и добавяме в контекста на транзакцията запис към таблицата **`Article`** в базата от данни **`Sales`**.

```
USE Sales

BEGIN TRANSACTION

INSERT INTO Article(Name) VALUES('*** Food ***')
```

Без да приключвате започнатата от предния скрипт транзакция, създаваме втора връзка към SQL Server (като стартираме втори път Query Analyzer или избираме от менюто File -> Connect). В прозореца на втората връзка изпълняваме следната команда, която се опитва да чете данни от таблицата **Article**:

```
USE Sales

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
SELECT Name FROM Article
```

Като резултат от изпълнението на горната заявка ще видим всички записи в таблицата **Article** плюс добавения (и все още непотвърден) от първата транзакция нов запис с име ***** Food *****. Този запис се избира от **SELECT** заявката защото нивото на изолация на текущата транзакция е настроено на **READ UNCOMMITED**.

Нека обаче да изпълним следния SQL скрипт в прозореца на втората връзка, който работи с повишено ниво на изолация на транзакцията на **SERIALIZABLE**:

```
USE Sales

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
SELECT Name FROM Article
```

Резултатът от изпълнението ще бъде "зависване" на изпълнението на командата **SELECT**. Това е така, защото нивото на текущата транзакция е зададено на **SERIALIZABLE**, а върху таблицата, която се опитваме да прочетем, има промени, започнали от паралелна незавършила транзакция.

Изпълнението на командата **SELECT** ще продължи, когато приключи първата транзакция, добавила нов запис в таблицата **Article**. За да се уверим в това можем да потвърдим първата транзакция, като изпълним от първия Query Analyzer командата:

```
COMMIT TRANSACTION
```

В този момент чакащата блокирала **SELECT** операция от втория Query Analyzer ще завърши успешно и ще покаже всички записи от таблицата **Article**, включително и новодобавения от другата транзакция.

Обърнете внимание, че нивото на изолация се задава по отношение на текущата транзакция, т. е. то контролира нейното поведение относно промени, нанесени от другите, изпълняващи се конкурентно транзакции, и не влияе на поведението на другите транзакции относно текущата. С други думи всяка транзакция чрез нивото си на изолация може да указва каква част от промените на другите транзакции иска да вижда и каква да изолира (и съответно да изчаква при необходимост).

При **READ UNCOMMITED** изолация текущата транзакция вижда всички започнали промени от всички транзакции, независимо дали са потвърдени или не. При изолация **SERIALIZABLE**, текущата транзакция вижда само потвърдени данни, като изчаква при необходимост завършването на започнатите от другите транзакции промени.

Пренасяне на база от данни

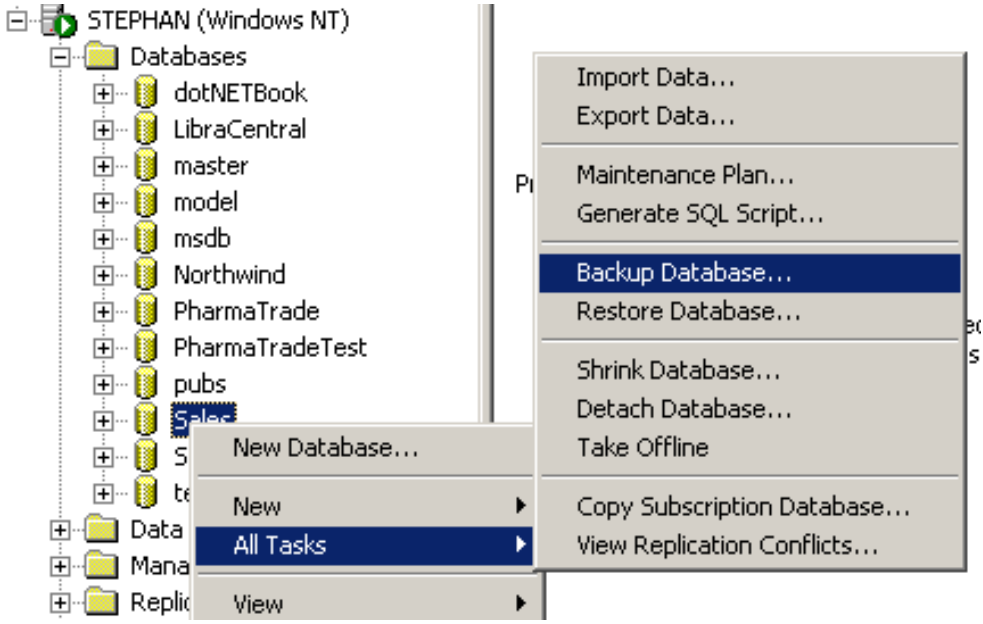
Сега ще разгледаме как можем да пренасяме вече създадена SQL Server база от данни, съдържаща таблици и данни в тях, от един компютър на друг. Тази операция често се налага при инсталация на дадено приложение при клиента, който ще го използва.

Пренасяне чрез архивиране и възстановяване

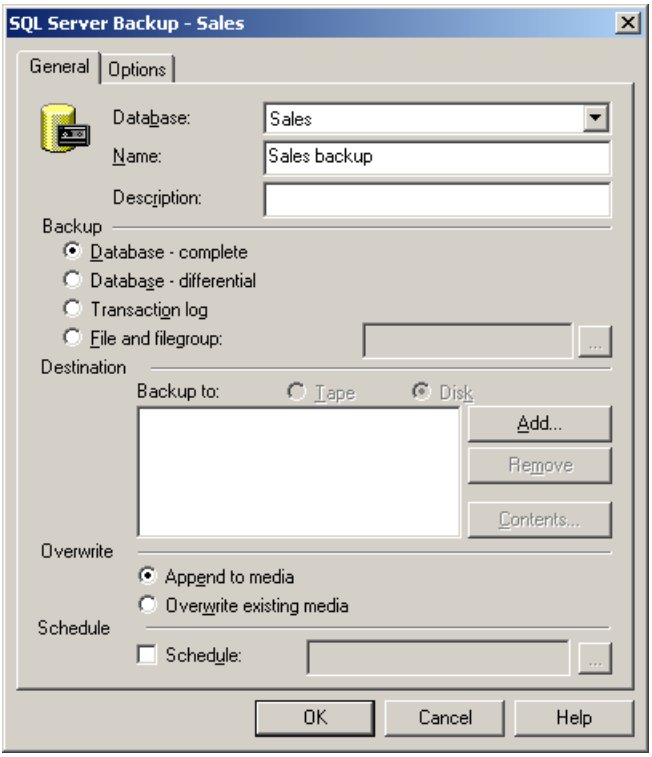
Една MS SQL Server база от данни може да се пренесе на друг компютър като се направи нейно архивно копие (backup) на сървъра-източник и след това на сървъра-приемник базата от данни се възстанови от това архивно копие.

За целта изпълняваме последователно следните стъпки:

1. От SQL Server Enterprise Manager избираме базата от данни и от контекстното меню избираме Backup Database, както е показано на картинката по-долу:

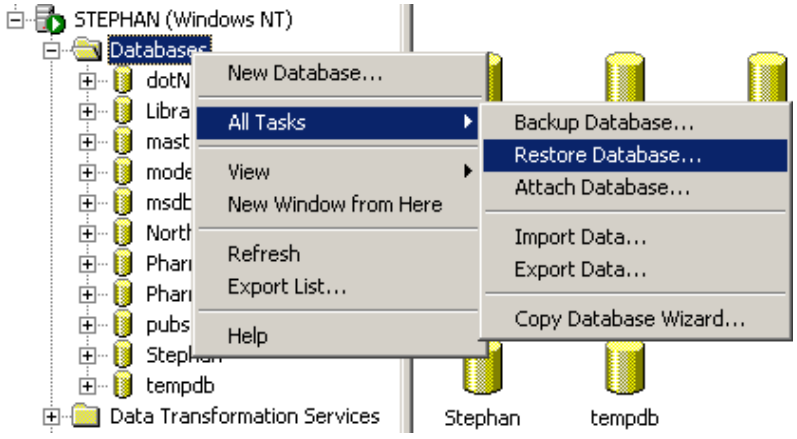


2. От диалога за определяне на параметрите при създаване на архива, задаваме името и местоположението на файла в който ще създадем архива. Това става чрез натискането на бутона [Add].

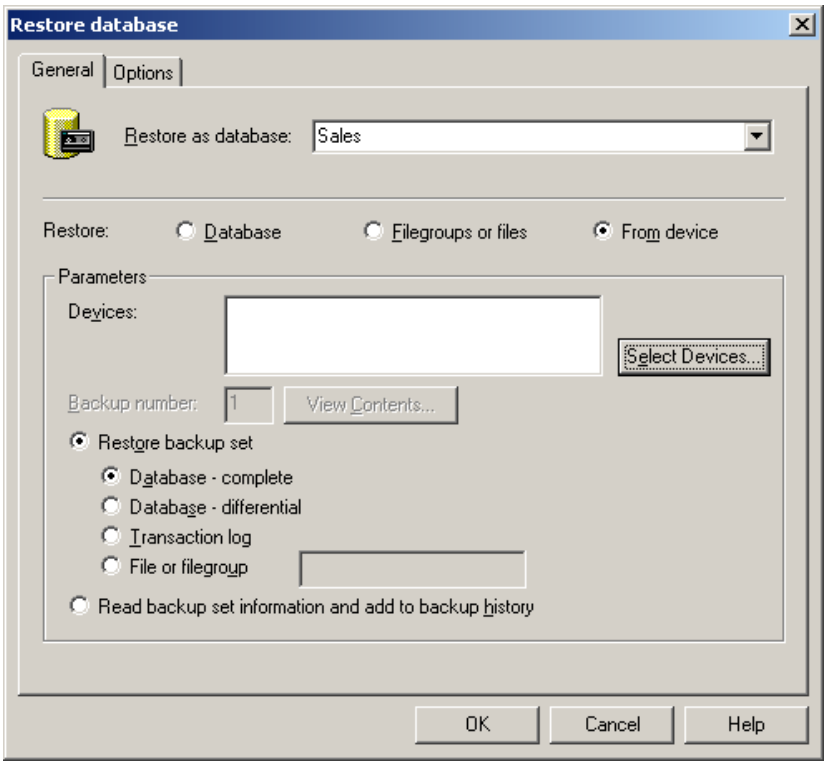


3. Натискаме бутона [OK], с което стартираме процеса на създаване на архивно копие. Сървърът ще създаде архивно копие на посочената база данни. Това става без да се спира нейната работа.

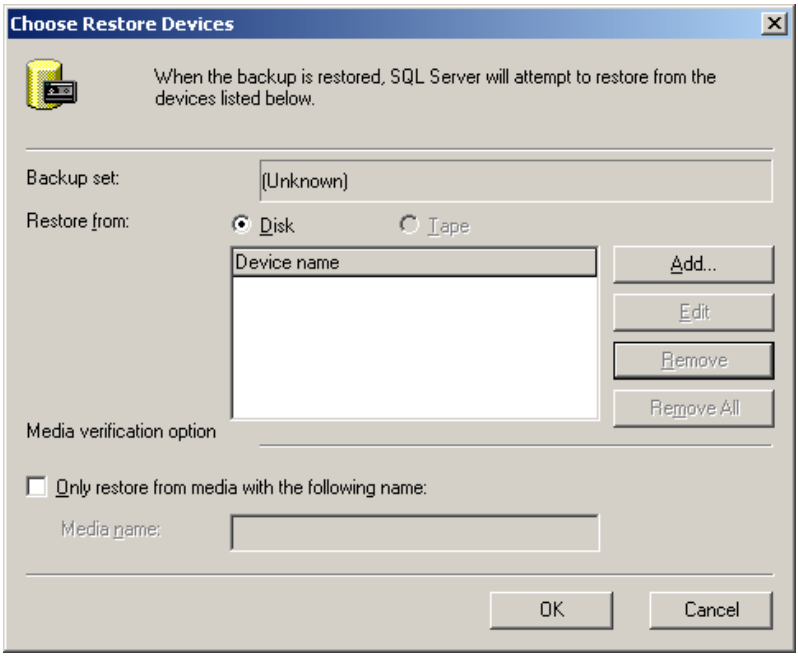
4. На сървъра-приемник избираме Restore Database, както е показано на картинката по-долу:



5. В появилия се диалог, показан по-долу, задаваме името, с което ще възстановим базата данни – в случая то е **`Sales`**. След това маркираме, че ще възстановяваме от устройство и избираме бутона [Select Device].



6. В появилият се диалог "Choose Restore Devices", избираме файла от който да възстановим базата от данни. Това е файлът, който сме създали на стъпка 2. Избирането на файла става чрез натискане на бутона [Add] и посочване на файла от появилия се диалог.



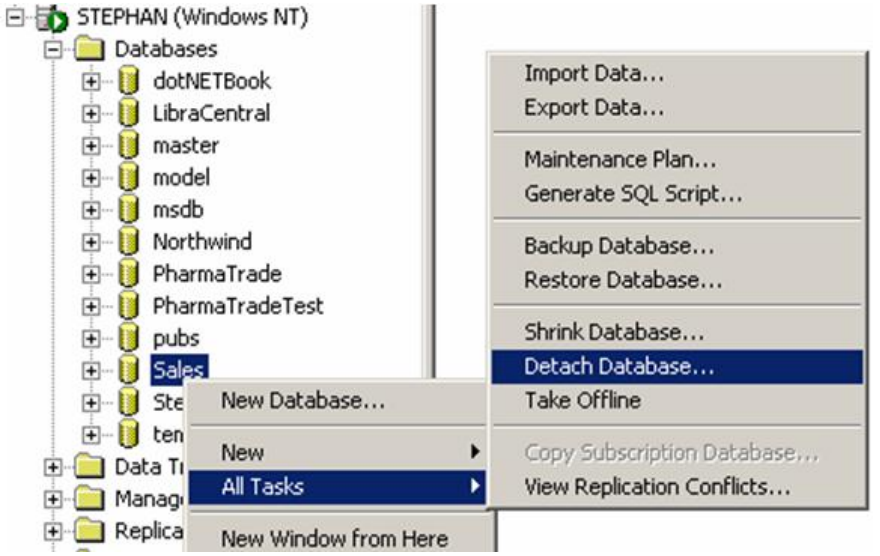
7. Избираме бутона [OK] в диалога "Choose Restore Devices", след което избираме същия бутон и от диалога "Restore Database". Това е последната стъпка, след която имаме вече възстановена база от данни.

Описаният метод за архивиране/възстановяване на база от данни, чрез използване на SQL Server Enterprise Manager е един от най-простите и лесни за употреба. За щастие той обикновено върши работа в голяма част от случаите. Освен него все пак тези операции може да се вършат по няколко други начина, например чрез използване на скрипт, написан на T-SQL. Това обикновено се прави, когато дадена база данни трябва да бъде инсталирана без намеса на потребителя.

Пренасяне чрез откачане и закачане

Пренасянето на база от данни може да се направи и чрез откачане на базата данни от сървъра-източник, копиране на файловете, от които тя се състои на сървъра-приемник и закачане на така копираните файлове на сървъра-приемник. За да реализираме този сценарии ще извършим следните действия:

1. От SQL Server Enterprise Manager избираме базата от данни и от контекстното меню избираме Detach Database, както е показано на картинката по-долу:

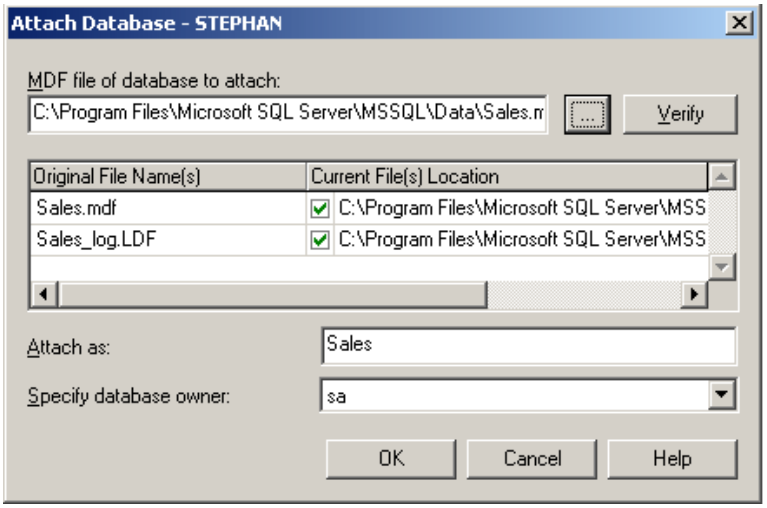


2. Потвърждаваме откачането на базата от данни в появилият се диалог Detach Database, чрез натискане на бутона [OK]. Ако той не е разрешен, това означава, че в момента има активни връзки към базата от данни. В този случай трябва да спрем всички приложения, които имат активна връзка към базата от данни или да натиснем бутона [Clear], който ще прекрати всички активни връзки.

3. Копираме файловете на базата от данни, на сървъра-приемник. Тези файлове се намират в Data директорията на SQL Server. Местополо-

жението на тази директория се указва при инсталиране на сървъра, като при обичайната инсталация тя се намира в поддиректорията на сървъра. Например местонахождението на файловете на базите от данни може да бъде: "**C:\Program Files\Microsoft SQL Server\MSSQL\Data**". Най-често базата от данни се състои от два файла: **<име_на_база>.mdf** и **<име_на_база_log>.ldf**. Администраторът на сървъра може да е задал базата от данни да се състои от няколко файла освен изброените по-горе, които да се намират на отделни устройства. Когато копираме файловете на базата от данни, трябва да се уверим че ще копираме всички нейни части.

4. Закачаме копираните от предната стъпка файлове на сървъра-приемник. За тази цел избираме групата Databases от Enterprise Manager и от контекстното меню избираме All Tasks → Attach Database. От показания на картинката по-долу диалог избираме **.mdf** файла на базата данни и натискаме бутона [OK].



Описаният по-горе метод за пренасяне на база данни чрез откачане и закачане работи единствено ако версиите на сървъра-източник на данните и сървъра-приемник са едни и същи.

Упражнения

1. Какви модели на базите от данни познавате?
2. Кои са основните функции, изпълнявани от една система за управление на бази от данни (СУБД)?
3. Дефинирайте понятието таблица в база от данни.
4. Обяснете разликите между първичен и външен ключ.
5. Посочете какви видове връзки между таблици познавате.

6. Кога дадена база от данни е нормализирана до четвърта нормална форма? Кои са предимствата на нормализираната база от данни?
7. За какво се използват ограниченията в една база от данни?
8. Проектирайте база от данни, съхраняваща данните за студентите, преподавателите и предметите, изучавани в един факултет.
9. Посочете предимствата и недостатъците на използването на индекси в базите от данни.
10. Какво е основното предназначение на езика SQL?
11. За какво се използват транзакциите? Дефинирайте техните отговорности и обяснете нивата им на изолация.
12. Посочете основните системни компоненти на MS SQL Server.
13. Избройте основните инструменти, които се използват при разработване на софтуер за SQL Server. За какво служи всеки от тях?
14. Кои команди спадат към DDL? Опишете тяхното действие.
15. Напишете скрипт, съдържащ DDL командите, необходими за създаване на базата от данни, обслужваща даден факултет в университет. Базата трябва да обхваща студентите, преподавателите, изучаваните предмети, учебните програми и оценките на всеки студент.
16. Обяснете действието на командите, спадащи към групата DML.
17. Какви видове съединения на таблици познавате?
18. Напишете заявка, която извлича всички изучавани предмети в университета, заедно със записалите ги студенти.
19. Каква е употребата на агрегиращите функции в езика SQL?
20. Напишете заявка, която извлича за всяка учебна група средния успех на студентите, които я съставят.
21. За какво се използват DBCC командите в SQL Server?
22. За какво се използват съхранените процедури? Посочете примери за използването им.
23. Напишете съхранена процедура, която добавя нов студент към даден курс в базата от данни, предназначена за обслужване на факултет.
24. Кои команди се използват за управлението на транзакциите в T-SQL?
25. Напишете съхранена процедура, която заменя даден изучаван предмет от студент с друг.
26. Напишете съхранена процедура, която добавя нов преподавател, който води нов учебен предмет и записва всички студенти от трети курс за този учебен предмет. Поредицата операции трябва да се изпълнява атомарно – или всички операции да се изпълнят успешно, или никоя от тях да не се изпълни.

27. Какво е поведението на вложените транзакции?
28. При какви случаи е възможно възникването на ситуацията "мъртва хватка"?
29. Какви са начините за пренасяне на база от данни на друг компютър? Избройте предимствата и недостатъците им.

Използвана литература

1. Бранимир Гюров, Светлин Наков, Стефан Захариев, Лазар Кирчев, Достъп до данни с ADO.NET – http://www.nakov.com/dotnet/lectures/Lecture-13-ADO.NET-v1.01.ppt
2. Rebecca Riordan, Designing Relational Database Systems, Microsoft Press, 1999, ISBN 0-7356-0634-X
3. Kalen Delaney, Inside SQL Server 2000, Microsoft Press, 2001, ISBN 0-7356-0998-5
4. Microsoft Corporation, Microsoft SQL Server Books Online

Глава 14. Достъп до данни с ADO.NET

Необходими знания

- Базови познания за .NET Framework
- Базови познания за езика C#
- Базови познания по XML технологиите
- Познания по релационни бази от данни
- Познания по езика SQL

Съдържание

- **Достъп до данни с ADO.NET**
  - Модели за работа с данните - свързан и несвързан
  - Еволюция на приложенията
  - Архитектура на ADO.NET
  - Доставчици на данни
  - Връзка с MS SQL Server
- **Реализация на свързан модел**
  - Класовете **`SqlConnection`**, **`SqlCommand`** и **`SqlDataReader`**
  - Параметрични заявки
  - Автоматично генериран първичен ключ
  - Използване на транзакции
  - Връзка с други бази от данни през OLE DB
  - Правилна работа с дати
  - Графични изображения в базата данни
- **Реализация на несвързан модел**
  - Класът **`DataSet`**, силно типизиран **`DataSet`**
  - Класовете **`DataTable`** и **`DataRelation`** и **`DataView`**
  - Използване на **`DataAdapter`** и **`CommandBuilder`**
  - Типичен сценарий за работа с данни в несвързана среда
  - Връзка между ADO.NET и XML
  - Сигурността при приложенията с бази от данни

В тази тема ...

В настоящата тема ще разгледаме подробно двата модела за достъп до данни, които са реализирани в ADO.NET – свързан и несвързан. Ще опишем програмния модел на ADO.NET, неговите компоненти и доставчиците на данни. Ще обясним кои класове се използват за свързан достъп до данните, и кои – за несвързан.

При разглеждането на свързания модел за достъп до данни ще се спрем на доставчика на данни **SqlClient** за връзка с MS SQL Server и ще обясним как се използват класовете **`SqlConnection`**, **`SqlCommand`** и **`SqlDataReader`**. Ще разгледаме работата с параметризирани заявки и използването на транзакции от ADO.NET. Ще дадем пример за достъп и до други бази от данни през OLE DB. Ще разгледаме и някои проблеми при работа с дати и съхранение на графични изображения в базата данни.

При разглеждането на несвързания модел за достъп до данни ще разгледаме в детайли основните ADO.NET класове за неговата реализация – **`DataSet`** и **`DataTable`**. Ще обясним как се използват ограничения, изрази, релации и изгледи в обектния модел **`DataSet`**. Ще се спрем подробно на класа **`DataAdapter`** и вариантите за неговото използване при зареждане на данни и обновяване на базата от данни. Ще разгледаме подходите за решаване на конфликти при нанасяне на промени в базата данни. Ще се спрем и върху връзката между ADO.NET и XML, а за финал ще разгледаме проблемите със сигурността в приложенията, използващи бази от данни.

Модели за работа с данни в ADO.NET

Преди да се запознаем в детайли с технологията ADO.NET, ще се спрем на моделите за достъп до данни, които се използват при изграждането на информационни системи. В практиката се използват свързан модел с постоянна връзка към базата данни и несвързан модел, при който връзката с базата данни не е постоянна и се осъществява за кратко – само за зареждане на данните и нанасяне на промените по тях.

Свързан модел (connected model)

Свързаният модел за работа с данни изисква постоянна връзка с базата от данни (online). При реализацията на този модел, която предлага ADO.NET, данните могат да бъдат четени само напред – връщане назад не е възможно. Промени не могат да се правят докато данните се четат – за тази цел трябва да се изпълняват отделни заявки към базата. Затова обикновено свързаният достъп се използва когато е необходимо да се прочетат данни, или да се направи единична промяна, но не се изисква сложна обработка на много информация от базата.

При други технологии, като например ADO, свързаният модел предлага възможност за четене на данните и в двете посоки, както и за извършване на промени чрез т. нар. updatable cursors. Основен проблем на този модел, както ще видим по-нататък, е лошата скалируемост.

При свързания модел достъпът до данните се осъществява в среда, в която винаги има връзка със сървъра на базата от данни. Това означава, че към базата има постоянно отворен TCP сокет или друг вид постоянна връзка като Named Pipe. Тя се отваря явно от приложението и също така явно би трябвало да се затвори. Докато връзката е отворена, в нея може да се създаде транзакция. Проблемът за управление на транзакциите ще разгледаме по-късно.

Следната схема изобразява свързания модел за достъп до на данни при ADO.NET:

В ADO.NET свързаният модел се реализира с **`Connection`**, **`Command`** и **`DataReader`** класове, които ще разгледаме подробно след малко.

Свързан модел – за и против

Нека разгледаме основните предимства и недостатъци на свързания модел, за да можем по-лесно да преценяваме кога е подходящо да го използваме и кога трябва да се спрем на друг подход.

Предимства

Свързаният модел работи директно със SQL заявки и е по-близък до релационните бази от данни. Това прави производителността му много добра. При него се изискват по-малко усилия от страна на разработчика.

Съществено предимство на този подход е, че по-лесно се контролира конкурентният достъп на много потребители до данните в базата. Има специални механизми, чрез които може да се гарантира целостността на данните. Освен това вероятността да се работи с текущата версия на данните е много по-голяма, отколкото при несвързания модел. Връзката с базата от данни е отворена през цялото време на работа и се предполага, че данните ще се извличат на по-малки части. Поради това, ако междувременно са настъпили някакви промени в данните, е много вероятно тези промени вече да са отразени в изтеглените данни.

Недостатъци

Един от основните недостатъци на свързания модел е необходимостта от постоянна мрежова връзка с източника на данните. В някои случаи това може да се окаже проблемно и би довело до голямо натоварване на мрежата. Това не е единственият проблем. При много организации връзката между отдалечени офиси и централния сървър не е надеждна и свързаността не е постоянна, поради което се налага offline работа.

Друг недостатък е заемането на ресурси (отворени връзки към базата) за продължително време. Резултат от това е лошата скалируемост на приложение, ползващо този модел – при нарастване на броя потребители може да се окаже, че физическите ресурси са недостатъчни да обслужат всички заявки. Проблемът е особено сериозен при Интернет приложенията, където е възможно едновременно много голям брой потребители да направят заявка към база от данни.

Свързан модел и транзакции

За да се осигури целостта на данните при конкурентен достъп от много потребители в свързания модел обикновено се използват транзакции.

След като се създаде една транзакция, в нейните рамки може да се изберат данните и да се осъществят необходимите промени върху тях. Така се подсигурява, че докато се извършва обработката данните ще бъдат предпазени от външни промени. В този случай не е необходимо да се полагат допълнителни усилия, за да се подсигурят данните при конкурентен достъп. Това се прави от самия сървър за бази данни.

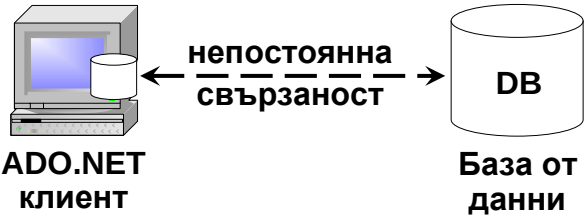
В някои случаи горното предимство може да се превърне в недостатък. Съществува опасност транзакционният достъп до данните едновременно от много потребители да доведе до нужда от изчакване на потребителите. Ако базата от данни реализира транзакциите чрез заключване на селектираните записи, транзакцията трябва да бъде потвърдена възможно най-бързо след първата промяна. В противен случай други потребители, които искат да извършват промени в същите записи, ще трябва да чакат нейното приключване. Това може значително да влоши производителността, ако транзакцията извършва продължителна обработка.

Несвързан модел (disconnected model)

При **несвързания модел** работата с данните се осъществява offline – данните се изтеглят от базата и се съхраняват на локалната машина. Първоначално се осъществява връзка към източника на данни, за да се извлече необходимото подмножество от данни. То се съхранява в локалната система и връзката се затваря. След това върху тях се извършва необходимата обработка. През това време няма връзка с базата (затова моделът се нарича несвързан). След извършване на нужните операции с данните връзката може да се отвори отново за да се въведат направените промени и/или да се извлекат още данни.

При несвързания модел транзакциите са много кратки – траят само докато се селектират или обновят данните без времето за тяхната обработка. В този случай не може една транзакция да започне преди извличане на данните и да продължи до евентуалната промяна. Този модел може да се използва при различни сценарии – например сложна обработка на данните, обработка на по-голям обем от данни или обработка на данни от различни източници.

Следната схема илюстрира несвързания модел за достъп до данните.



В ADO.NET несвързаният модел се реализира с `DataSet` и `DataAdapter` класове, които ще разгледаме подробно малко по-късно.

Примери за използване на несвързан модел

Несвързаният модел може да се приложи при достъп до данни чрез уеб услуга. В този случай услугата извлича данните от източника и ги предава като несвързан обект към клиента. Клиентът ги обработва и след известно време изпраща до услугата направените от него промени, а услугата ги нанася в базата данни.

Друг пример за приложение на несвързания модел е за интеграция с XML. Данните могат да се извлекат в несвързан обект и от него да се съхранят в XML формат. По-късно съхранените XML данни могат отново да станат несвързани обекти. ADO.NET предлага удобни средства за извършване на тези операции, както ще видим в края на настоящата тема.

Несвързан модел – за и против

Сега ще разгледаме основните предимства и недостатъци на несвързания модел, за да знаем в какви ситуации да го използваме и в кои да го избягваме.

Предимства

Основно предимство на несвързания модел е, че клиентът се свързва с основната база от данни само когато има нужда. През останалото време той работи без връзка към нея. Това намалява натоварването на сървъра на базата от данни и изразходва по-малко ресурси.

Често се случва приложението да се нуждае само да чете и визуализира данните. Тези операции могат да бъдат извършени и в несвързан режим, като при това не се заемат ресурси на сървъра и други потребители също могат да се свързват междувременно.

При намаляване на броя отворени връзки към базата се подобрява значително скалируемостта на приложението. Увеличаването на броя потребители не води до много по-голямо натоварване на базата от данни, защото ресурсите за обслужване на всеки потребител са необходими само докато трая неговата заявка и след това се освобождават.

Друго предимство на несвързания модел е възможността приложението да работи в offline режим, без да има постоянна физическа свързаност със сървъра на базата данни.

Недостатъци

Недостатък на несвързания модел на достъп е, че данните при клиента не винаги са текущи. След изтегляне и прекъсване на връзката с базата, в нея данните могат да бъдат променени от други потребители, но тези промени няма да се отразят върху изтеглените данни. Така има опасност да се работи с остарели данни.

Друг недостатък е, че трябва да се положат допълнителни усилия от разработчика за разрешаване на конфликтите между различните версии на данните. Подобен случай възниква, ако след запазване на данните на локалната машина потребителят промени някаква стойност в тях, а същевременно друг потребител промени същата стойност в основната база. Тогава, когато потребителят се опита да обнови данните в базата, настъпва конфликт. На проблема с разрешаване на конфликтите при обновяване на данни в несвързан модел ще обърнем специално внимание малко по-късно.

Еволюция на приложенията

Да разгледаме основните типове софтуерни архитектури, които се използват за приложения работещи с бази от данни. Ще ги класифицираме според броя слоеве. Слоевете представляват логическо разделяне на функционалността на приложението. Типовете приложения са разгледани в нарастваща сложност, като това е и последователността, в която исторически са възникнали различните архитектури.

Еднослойни приложения

Еднослойните приложения представят най-простия тип архитектура. При тях няма логическо (или физическо) разделяне на функционалността в отделни слоеве – цялото приложение е едно цяло. Тези приложения обикновено се използват само от един потребител и затова са сравнително прости за реализация.

Като предимство на еднослойните приложения може да се посочи това, че всички техни компоненти се намират на едно място, което улеснява проектирането и изграждането на приложението. От друга страна именно това е причина при промяна във функционалността на системата да се налага пълна преинсталация, но това не е най-големият проблем на тази архитектура. Еднослойните приложения много трудно могат да се използват едновременно от много потребители и това силно ограничава приложението им в практиката.

Като пример за еднослойно приложение можем да посочим просто настолно (desktop) приложение, използващо вградена в него локална база от данни и предназначено за работа на един компютър. При него не се очаква нито голямо натоварване, нито необходимост от обработка на големи обеми от данни, нито достъп от много потребители едновременно.

Такова приложение е, например, Microsoft Excel. Той оперира локално на един компютър, работи с локални данни, но не позволява дадена таблица да се променя едновременно от няколко потребители.

Двуслойни приложения (клиент-сървър)

Клиент-сървър е широко разпространена софтуерна архитектура. При този модел приложението логически се разделя на клиент и сървър, като клиентът отправя заявки към сървъра, а сървърът ги изпълнява и връща резултат. Този модел често се използва при по-прости разпределени и мрежови приложения.

Клиентът и сървърът се разглеждат като два отделни слоя на приложението. В клиента се разполагат потребителският интерфейс и бизнес логиката на системата. Този слой е отговорен за представяне на информацията и взаимодействие с потребителя, както и за извършване на необходимата обработка според логиката на приложението. Във втория слой (сървъра) се съхраняват данните. Той отговаря за тяхното управление и

съответно за операциите с базата от данни. Не е задължително двата слоя да са физически разделени (на различни машини), макар че най-често случаят е точно такъв.

Предимства на модела клиент-сървър

Основното предимство на клиент-сървър архитектурата е разделянето на функционалността между двата слоя. Така се избягва прекомерното натоварване на сървъра при обслужване на множество заявки, тъй като основната част от обработката не се извършва от него, а от клиентите.

Разделянето на клиент и сървър позволява много потребители (клиенти) да бъдат обслужвани от сървъра едновременно, а това е необходимо във всички по-сериозни информационни системи.

Клиент-сървър моделът има и друго предимство. Понеже логиката е при клиента, той не трябва непрекъснато да комуникира със сървъра при извършването на обработката, а това намалява мрежовия трафик. В някои случаи това може да се окаже недостатък, защото натоварването на клиента с изпълнението на бизнес логика не винаги е добра практика.

Недостатъци на двуслойните приложения

При тази архитектура поддръжката на приложенията се усложнява значително. При промяна в логиката на приложението се налага преинсталация на клиентската част, а при голям брой потребители това би означавало да се прави преинсталация на приложението на голям брой компютри, което може да доведе до значителни трудности и големи разходи.

Всъщност по-големият проблем при тази архитектура е друг. Един сървър не е в състояние да обслужва едновременно прекалено много потребители, а това води до лоша скалируемост. За този проблем решение може да бъде кластеризацията на сървъра, но при големи системи обикновено се ползва архитектура с повече слоеве.

Примери за двуслойни приложения

Типичен пример за двуслойни приложения са повечето системи за управление на склад. При тях складовите наличности се съхраняват на сървъра в база от данни, а обслужващият персонал към склада осъществява достъп до данните чрез специален клиентски софтуер. Възможно е в един склад да работят конкурентно няколко клиента към базата данни. Работата с актуални данни е много важна и затова при такива приложения обикновено се ползва свързан модел.

Като друг пример за клиент-сървър система може да се разгледат MS SQL Server и MS Query Analyzer. MS Query Analyzer представлява клиентската част, през която се отправят заявки за извличане или промяна на информация в базата от данни. Тези заявки се изпълняват от сървъра на базата – MS SQL Server. Резултатът от заявката се представя на потребителя от MS Query Analyzer.

Трислойни приложения

В едно приложение могат да се обособят три типа функционалност:

- представяне на информация на потребителя и взаимодействие с него
- обработка, която реализира логиката на приложението
- управление на данните

При двуслойните приложения тези три типа функционалност се поставят в два слоя, от което произтичат различни проблеми. Затова алтернатива на двуслойната архитектура е трислойната, при която трите типа функционалност са разделени съответно в три логически слоя – презентационнен слой (front-end), бизнес слой (бизнес логика) и съхранение на данните (back-end).

Презентационният слой (front-end) е потребителският интерфейс на приложението. Той е отговорен за взаимодействието с потребителя. В него не се реализира обработка на данни (бизнес логика). Единствената логика, което се използва, е презентационната – подготовка на данни за визуализация, валидация на данни, взаимодействие с потребителя и т.н. Презентационният слой не комуникира директно с базата данни. Вместо това той използва услугите на бизнес слоя.

В **бизнес слоят (business tier)** се обособява бизнес логиката на системата – в него се реализират работните процеси и се извършва цялата обработка на данните на приложението. Бизнес слоят се грижи за правилната работа с много потребители едновременно. Той комуникира с базата данни за да съхранява и обработва данните в нея.

Слоят за съхранение на данни (back-end) е отговорен за достъпа и съхранението на данните, като той извършва операциите с базата от данни. Най-често това е самият RDBMS сървър, например MS SQL Server. Слоят за данни не имплементира бизнес логика (работни процеси от системата), но може да имплементира логика, свързана с обработката на данните в самата база данни. Достъпът до слоя за данни става единствено от бизнес слоя.

Предимства на трислойния модел

Основното предимство на този модел е разделянето на трите типа функционалност. Това обособяване позволява промяна в някой от трите слоя да не окаже влияние върху останалите. Например, ако се промени бизнес логиката, изменение ще се наложи само във втория слой, като това не трябва да се отрази на работата с данните или представянето на информацията на потребителя.

Друго предимство е по-голямата скалируемост на този модел в сравнение с двуслойния и по-малкото натоварване на клиента, тъй като той само визуализира данните, а логиката е изнесена в бизнес слоя и не е при него. Изнасянето на логиката дава възможност за лесно отстраняване на

проблеми и обновяване само на бизнес слоя без да се променя клиента. Клиентът може да е най-обикновен уеб браузър.

Всеки от трите слоя може да се разположи физически върху отделна машина и така да се намали натоварването на всяка от машините, с което да се увеличи производителността при голям брой клиенти. Например, при една Интернет банкова система потребителската база данни на банката представя back-end слоя, уеб сървър изпълнява бизнес логиката и част от презентационната логика, според която се извършват плащания, финансови трансфери и т. н., докато компютърът на потребителя с уеб браузър предоставя потребителския интерфейс на системата.

Възможно е някой от слоевете физически да е разположен върху няколко машини, които работят в заедно клъстер. Това дава възможност за балансиране на натоварването, репликация на данни и процеси и често се използва при изграждането на скалируеми, високонадеждни отказоустойчиви приложения, предназначени да работят с голям брой потребители.

Друго предимство на трислойния модел е възможността да имаме различни типове клиенти, използващи един и същ бизнес слой. Например едно приложение може да е достъпно едновременно през настолен клиент и през уеб среда. При такъв сценарий ако не използваме трислоен модел, ще се наложи да се дублира бизнес логиката на приложението при два функционално еквивалентни клиента, а това е много лоша практика.

Недостатъци на трислойния модел

Обикновено функционалността на отделните слоеве (без потребителския интерфейс) се изпълнява от различни сървъри. Например бизнес логиката може да е на сървър за приложения (application server), а съхранението и достъпът до данните да се извършва от сървъра на базата от данни. Резултат от това е повишаването на риска за сигурността на системата, тъй като трябва да се подсигурят повече сървъри срещу евентуални атаки. Това води и до по-трудна поддръжка на този тип приложения.

Друг проблем на трислойната архитектура е намаляването на производителността заради нуждата от комуникация между слоевете.

Проблем е също и сложността на системата – трябва да се осигурят надеждни механизми за комуникация между слоевете, трябва да се предвидят проблемите при изчезване на свързаността между тях, да се отдели повече време за проектиране и имплементация и т. н.

Поради описаните проблеми трислойната архитектура се препоръчва за по-големи и сложни приложения, а не за всички.

Пример за трислойно приложение

Добър пример за трислойно приложение е една банкова система. В нея данните за сметките се съхраняват в база от данни (back-end). Логиката по управление на финансите (олихвяване, оценка на риска при отпускане на кредит, обслужване на сметки и т.н.) се разполагат в бизнес слоя.

Като презентационен слой (front-end) системата може да предоставя няколко приложения, например настолна система, от която работят служителите на банката, уеб система за Интернет банкиране и WAP система за банкиране от мобилни телефони.

В .NET за изграждане на трислойни архитектури най-често се използва следният модел: back-end система за съхранение на данните (например MS SQL Server), ASP.NET уеб услуга за реализация на бизнес логиката и различни клиенти – Windows Forms GUI приложение и ASP.NET уеб приложение за реализация на front-end частта.

Многослойни приложения

В някои случаи се оказва подходящо трислойната архитектура допълнително да се разшири. Така се получават системи с четири и повече логически слоя. Това дава възможност да се разширява функционалността на системата без да се променят всички слоеве. Възможно е различните слоеве да работят върху различни машини и така да се намали натоварването на отделните машини.

Възможно е, например, да се раздели бизнес слоят на два слоя – единият да е ориентиран към обработка, свързана с потребителския интерфейс, а другият да отговаря за интегриране и манипулиране на данните. Друг сценарий е интегрирането на данни от множество източници. В такъв случай може да се добави допълнителен сървър между сървъра на приложението и сървърите на базите от данни, който се грижи да представи разпределените данни на сървъра на приложението така, сякаш са от една база.

Понякога се налага използване на многослойна архитектура заради хетерогенността на средата – отделните слоеве са реализирани върху различни платформи и с различни технологии, например на C++ върху Linux, Java върху Solaris и .NET върху Windows.

Предимства на многослойните приложения

Разделянето на функционалността дава допълнителна гъвкавост на приложенията. Повишава се тяхната скалируемост. Става възможно различни приложения да достъпват части от функционалността на системата през отворени протоколи.

Недостатъци на многослойните приложения

Поради усложнената архитектура на многослойните системи много трудно се дефинират и реализират правилата за сигурен достъп. В тези системи има много повече компоненти, които трябва да бъдат защитени. Друго следствие от усложнената архитектура е по-трудната разработка на такива системи. Изисква се повече време за тяхното планиране и разработка.

В заключение можем да кажем, че с увеличаване на броя на слоевете на приложението се увеличава неговата скалируемост, гъвкавост и неговите

възможности, но пропорционално на това нараства и сложността. Така че винаги трябва да се търси баланса между тези параметри. Окончателният избор зависи от конкретните изисквания на разработваната система.

Какво е ADO.NET?

След като направихме преглед на различните модели и архитектури за изграждане на приложения с бази от данни, е време да се заемем с ADO.NET – технологията, която .NET Framework предоставя за достъп до данни.

ADO.NET представлява набор от библиотеки за работа с данни, включени в .NET Framework. Те включват класове, интерфейси, структури и други типове и са предназначени за достъп до различни източници на данни. ADO.NET е изцяло базиран на .NET Framework и притежава много от неговите характеристики – поддръжка на множество езици, автоматично управление на паметта, обектно-ориентиран дизайн, обща система от типове и конвенция за именуване. Предоставят се средства, които позволяват с данните да се работи независимо от това от какъв източник идват.

ADO.NET предлага програмен модел за работа с данните, който съответства на двата модела за достъп до данни – свързан и несвързан. Освен това обектният модел на ADO.NET предлага много фин контрол върху връзката с източника, изпълнението на команди и обработката на данните. В ADO.NET се прави ясно разграничаване между достъпа до данните и тяхната манипулация.

Доскоро широко използвана Windows технология за достъп до данни беше ADO – ActiveX Data Objects. ADO.NET е неин наследник, но е нещо много повече от нейна нова версия. Разширява я с много нови възможности, които се налагат от развитието на технологиите. Съобразява се с изпълнението в Интернет среда, която налага да се използва достъп с непостоянна връзка (connectionless).

Важна характеристика на ADO.NET е възможността за работа в несвързана среда. Това е и една от основните разлики с ADO. В ADO.NET тази възможност е вградена, а това улеснява разработката на многослойни приложения и уеб услуги.

Вградената поддръжка на работа с XML е друго голямо предимство на ADO.NET спрямо ADO. ADO.NET предлага разнообразни средства за осъществяване на връзка с XML. Това значително подпомага изграждането на приложения, които са независими от източника на данни, с който работят, и които взаимодействат помежду си.

Пространства от имена на ADO.NET

Различните класове и интерфейси, които предлага ADO.NET, са разпределени в няколко основни пространства от имена:

- **System.Data** – съдържа основните архитектурни класове на ADO.NET. В него влизат например класовете **DataSet**, **DataTable**, **DataRow**, които ще разгледаме подробно малко по-късно, както и много други класове.
- **System.Data.Common** – в това пространство от имена се съдържат класове, които се използват независимо от източниците на данни, като например **DataAdapter**.
- **System.Data.SqlClient** – включва специфични класове за връзка със SQL Server, които позволяват да се осъществи връзка с MS SQL Server, да се извличат данни и да се изпълняват команди. Някои от класовете в това пространство от имена са **SqlConnection**, **SqlCommand**, **SqlDataReader** и др.
- **System.Data.SqlTypes** – съдържа класове които съответстват на вградените в SQL Server типове данни и представляват по-бърза и сигурна алтернатива на другите типове. Включва **SqlInt32**, **SqlDouble**, **SqlDateTime** и много други.
- **System.Data.OleDb** – осигурява класове за връзка с OleDB източник на данни. В него влизат например класовете **OleDbConnection**, **OleDbCommand** , **OleDbDataReader** и др.
- **System.Data.Odbc** – класове за връзка с ODBC. Съдържа например **OdbcConnection**, **OdbcCommand** и др.
- **System.Xml** – това пространство от имена съдържа класове, които поддържат обработката на XML данни и връзката между релационния модел и XML. Често използвани са например класовете **XmlDocument** и **XmlDataDocument**.

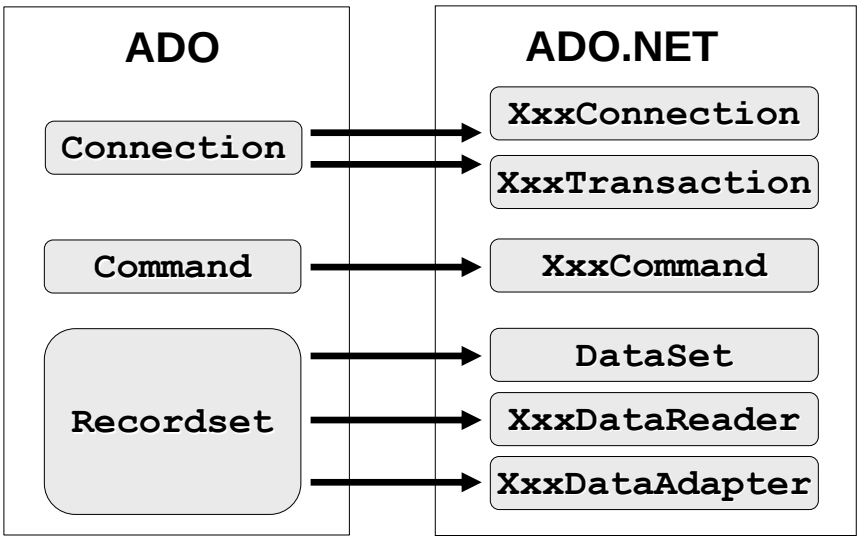
Еволюция на ADO към ADO.NET

ADO.NET е ориентирано към два основни сценария на обработка, които отразяват двата модела за достъп до данни (свързан и несвързан) и съответно предоставя различни класове за тяхната реализация (основно **DataSet** и **DataReader**, които ще разгледаме след малко).

ADO (ActiveX Data Objects) е технология в Microsoft Windows, която предоставя единен стандарт за достъп до релационни бази от данни от Windows приложения посредством ActiveX обекти.

Счита се, че ADO.NET е следващата версия на ADO, неговият наследник, но различията между двете технологии са сериозни и за някои неща няма аналогия.

Следната илюстрация показва най-общо връзката между основните класове в ADO и ADO.NET:



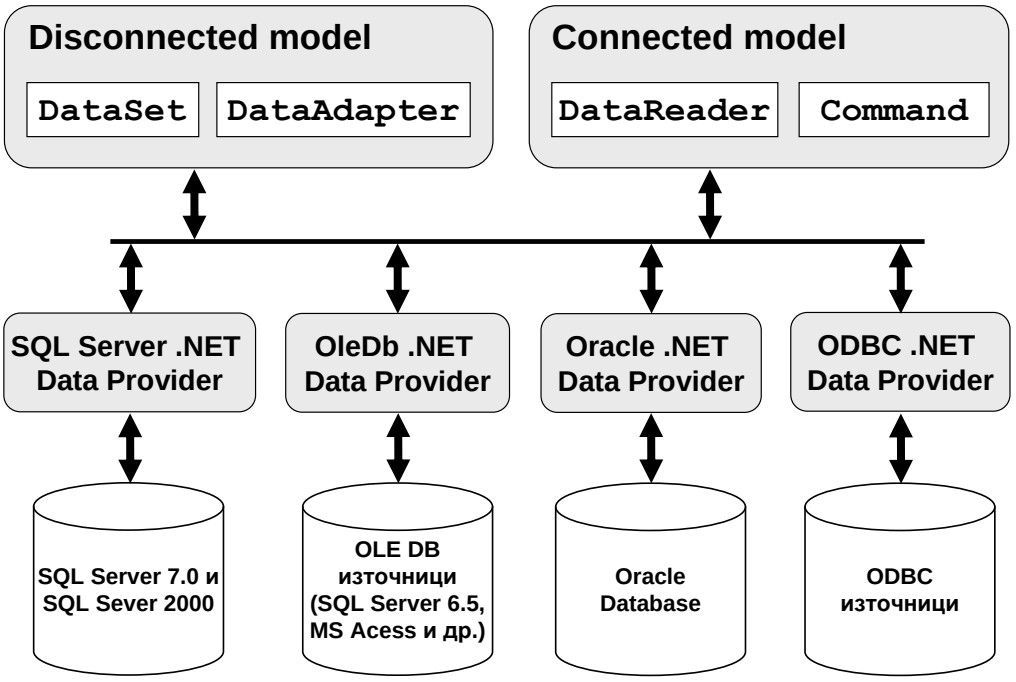
В старата ADO технология за извличане на данни се използваше единствено **Recordset**. Един **Recordset** представлява редактируема таблица (върху данните могат да се правят промени, които се отразяват в базата) и позволява навигация и в двете посоки. Последните две възможности не се предоставят от **DataReader** класовете, чрез които е реализиран свързаният модел в ADO.NET.

Друга характерна особеност е, че ADO.NET не предоставя единен обектен модел (както е в ADO), а вместо това осигурява специфични класове, в зависимост от това какъв начин за връзка към базата се използва. Затова например, докато в ADO има единствено **Connection**, в ADO.NET има **SqlConnection**, **OleDbConnection** и др. Освен това, ADO.NET добавя самостоятелен клас, представящ транзакциите, докато в ADO тази функционалност се реализира чрез класа **Connection**.

Компоненти на ADO.NET

ADO.NET се състои от компоненти за работа в свързана и несвързана среда, които могат да имат имплементация за различни бази от данни – както релационни, така и други, например дървовидни или XML базирани. За достъп до различните бази от данни се използват т. нар. доставчици на данни (data providers). Те са специфични за съответната база, но спазват програмния модел на ADO.NET като имплементират дефинираните в него интерфейси.

Диаграмата по-долу изобразява основните компоненти на ADO.NET:



В следващите секции ще се спрем подробно на всеки от тях.

Доставчици на данни (Data Providers) в ADO.NET

Доставчиците на данни (Data Providers) са съвкупности от класове, които осъществяват връзка с различни бази от данни. Те осигуряват възможност да се изпълняват команди и да се получават резултатите по начин, който е независим от източника на данни и неговата специфична функционалност. Те създават тънък слой между ADO.NET приложението и източника на данните (базата данни). Доставчиците на данни са проектирани да осигуряват ефективен достъп за промяна на данните или само за извличане.

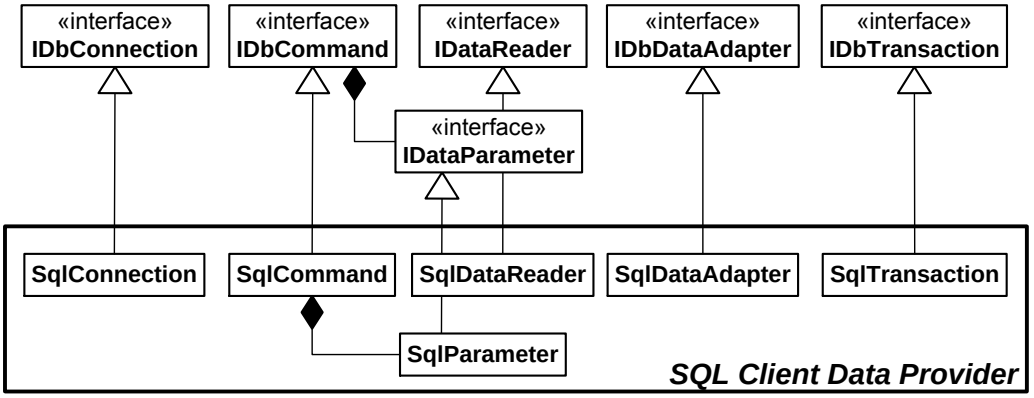
За различните RDBMS се използват различни доставчици (Data Providers), като всеки е оптимизиран за работа със съответната база от данни. Това се налага, тъй като различните производители използват различни протоколи за достъп до сървъра на базата. Различните доставчици на данни осигуряват сходна базова съвкупност от възможности, но въпреки това е възможно между тях да има разлика във функционалността. Това се дължи на разликите между различните източници, както и на разлики в имплементацията.

Класове на един Data Provider

Всеки доставчик на данни (Data Provider) съдържа 4 основни класа, с помощта на които се осъществява достъпа до съответната база от данни. Те осигуряват връзка към базата (**`Connection`**), изпълнение на команди (**`Command`**), поточно извличане на данни (**`DataReader`**) и адаптери за работа при несвързан модел (**`DataAdapter`**). Допълнително в доставчиците

на данни са реализирани и класове за работа с параметрични заявки (**Parameter**), за работа с транзакции (**Transaction**) и др.

С цел унифициране на класовете за работа с релационни бази от данни, реализирани от различните доставчици на данни в ADO.NET са дефинирани някои общи интерфейси, които се имплементират от доставчиците по специфичен за тях начин. Това са: **IDbConnection**, **IDbCommand**, **IDataReader**, **IDbDataAdapter**, **IDataParameter**, **IDbTransaction** др. Следната клас-диаграма онагледява връзката между тези интерфейси и доставчиците на данни:



На диаграмата са изобразени основните интерфейси от програмния модел за доставчиците на данни от ADO.NET и съответните класове, които ги имплементират в SQL Client Data Provider.

Класове за връзка (Connection)

Класът **Connection** осъществява връзката към базата от данни. Обикновено се създава по Connection String, който задава параметрите на връзката. **Connection** класът предлага методи **Open()** и **Close()**, с които се отваря и затваря връзката. По **Connection** се създават **Command** обекти.

Класове за изпълнение на команди (Command)

Класът **Command** представя заявка към база от данни. **Command** има свойство **CommandText**, което съдържа SQL заявка. Ако не се очаква изпълнението на заявката да върне резултат (например в случай на **INSERT** или **DELETE**), тя се изпълнява с метода **ExecuteNonQuery**(). Ако резултатът е само една стойност може да се извика **ExecuteScalar**(), а ако резултатът е таблица (един или повече редове), заявката се изпълнява с метода **ExecuteReader**(), който връща **DataReader** обект.

Класове за извличане на данни (DataReader)

Класът **DataReader** се използва за извличане на данни. Той осигурява последователен достъп до данните от една или повече таблици. Реализира се чрез курсор на сървъра и поради това докато се чете е необходимо

съответната връзка да е постоянно отворена. При извличане на данни с **DataReader** достъпът до отделните редове с данни се осъществява последователно, еднопосочно (forward only) и само за четене (read only).

Класове за работа с транзакции (Transaction)

За работа с транзакции е предвиден класът **Transaction**. Транзакция може да се стартира по два начина. Единият е чрез извикване на метода **BeginTransaction()** на **Connection** обекта, като методът връща обекта на стартираната транзакция. Друг начин е да се присвои обект от клас **Transaction** на свойството **Transaction** на **Command** обект (в този случай двата обекта трябва да са прикачени към един и същ обект за връзка с базата). Класът **Transaction** има методи **Commit**() (за потвърждаване на транзакция) и **Rollback**() (за отмяна на транзакция). По-нататък ще се спрем по-подробно върху работата с транзакции.

Класове за връзка с несвързани данни (DataAdapter)

Класът **DataAdapter** се използва за връзка между несвързаните компоненти (като **DataSet** и **DataTable**) и базата от данни. Той прочита данни от базата в **DataSet** или записва данните от **DataSet** в базата.

Стандартни доставчици на данни в ADO.NET

В ADO.NET (версия 1.1) има следните стандартни доставчици на данни:

- **SqlClient** – той е предназначен за връзка с MS SQL Server (версия 7.0 или по-късна). Този доставчик на данни е оптимизиран за работа със SQL Server – използва собствен протокол (TDS) за комуникация със сървъра и се свързва директно с него, а не минава през междинен OLE DB или ODBC слой.
- **OleDb** – този доставчик на данни се използва за връзка с бази от данни, които поддържат стандарта OLE DB. Повечето известни сървъри за управление на бази от данни (например Oracle, DB2, SQL Server, MySql, Interbase, PostgreSql и др.) имат OleDB драйвери и могат да бъдат използвани чрез OleDb доставчика.
- **Odbc** – използва се за връзка база от данни по стандарта ODBC. Поддържат се всички по-известни database сървъри. OLE DB и ODBC по принцип са конкуриращи се стандарти, но OLE DB е по-съвременния от двата и трябва да се предпочита пред ODBC, защото осигурява по-голяма гъвкавост, по-висока надеждност и има по-добра поддръжка.
- **Oracle** – предназначен е за връзка с Oracle източници на данни.

Освен стандартните доставчици на данни, които са вградени в ADO.NET, съществуват и доставчици за директна връзка с други RDBMS, които се предлагат от трети производители. Такива са налични за:

- IBM DB2

- MySQL
- PostgereSQL
- Borland Interbase / Firebird

За връзка с бази от данни, които не се поддържат стандартно от .NET Framework, се препоръчва да се използват специфичните за тях .NET Data Providers, а не да се ползва OLE DB или ODBC, защото това намалява производителността и ограничава достъпната функционалност, специфична за съответната база от данни.

Компоненти за работа в несвързана среда

Освен доставчиците на данни ADO.NET съдържа и независими от доставчика компоненти за работа в несвързана среда – **`DataSet`**, **`DataTable`**, **`DataRelation`**, **`DataView`** и други. Тяхното основно предназначение е да съхраняват данни в релационен вид в паметта и да позволяват тяхната обработка. Свързания модел на работа в ADO.NET ще дискутираме в детайли малко по-нататък.

SqlClient Data Provider

До сега разгледахме архитектурата на ADO.NET. Нека сега се запознаем подробно с възможностите, които предлага **SqlClient Data Provider**.

Както беше отбелязано, той е оптимизиран за работа с MS SQL Server (осъществява директна връзка със сървъра) и може да се използва за версия 7.0 и по-нови (например SQL Server 2000 и SQL Server 2005). Класовете на този доставчик на данни се намират в пространството от имена **`System.Data.SqlClient`** и започват с префикса **`Sql`** (например **`SqlConnection`**, **`SqlCommand`** и др.).

Основните класове на **`SqlClient`** Data Provider са:

- **`SqlConnection`** –осъществява връзката с MS SQL Server. Предоставя методи за отваряне и затваряне на връзка, за започване на транзакции и др. При използването на този клас програмистът не се интересува от детайлите за това как точно се осъществява физическата връзка с базата.
- **`SqlCommand`** – изпълнява команди върху MS SQL Server през вече установена връзка. Той обвива текст на SQL заявка или извикване на съхранена процедура.
- **`SqlDataReader`** – този клас служи за извличане на данните от SQL сървъра. Обект от него се създава имплицитно в резултат от изпълнение на команда.
- **`SqlTransaction`** – използва се за работа с транзакции. Има методи **`Commit()`** и **`Rollback()`** за потвърждаване и отмяна на транзакции, съответно.

- **`SqlDataAdapter`** – служи за "мост" между **`DataSet`** обект и SQL Server – през него се запълва **`DataSet`** и се обновява базата. Той сам управлява връзката с базата от данни – сам я отваря и затваря, като програмистът може да не се грижи за това.

Видове автентикация в SQL Server 2000

SQL Server предлага два механизма за автентикация на потребителите – Windows автентикация и смесена.

Windows автентикация

Windows автентикацията се базира на модела за сигурност в Windows NT 4.0 и Windows 2000 и се използва по подразбиране от SQL Server. При нея се разчита на автентикация на потребителя от операционната система. В този случай право на достъп се дава на Windows потребители и групи (или потребители от Microsoft Active Directory при работа в домейн). Когато се използва този модел на сигурност, администраторът на базата от данни дава достъп на потребителите до компютъра, на който работи базата и им дава права за влизане в SQL Server 2000.

Този метод на автентикация осигурява сигурна валидация и криптиране на различни нива – криптиране на данните, на информацията, обменена при автентикация на потребител и др. Той предлага одитиране на достъпа, за което разполага с вграден механизъм за извършване на одит. Този механизъм позволява проследяване на употребата на правата в SQL Server 2000. Сървърът позволява и заключване на потребителски акаунти за забраняване на достъпа до базата на съответния потребител.

Смесена автентикация

При този метод потребителите могат да бъдат автентикирани от Windows или от SQL Server 2000. Ако потребителят не може да използва стандартна Windows автентикация, автентикацията се извършва от SQL Server 2000, като за тази цел сървърът пази двойките име и парола на потребителите. Този модел се използва предимно за съвместимост с по-стари версии и в бъдещи версии на SQL Server може да спре да го поддържа. Има малка разлика в скоростта при двата модела на автентикация, като Windows автентикацията е по-бърза. Смесената автентикация е подходяща за хетерогенна среда, където не е наличен механизмът на Windows за автентикация.

Символен низ за връзка към база от данни (Connection String)

При създаване на връзка към база от данни е нужно да се зададат някои важни параметри, като например методът на автентикация, адресът на сървъра, името на базата и др. За целта се използва свойството **`ConnectionString`** на класа на връзката. То съдържа двойки име/стойност,

които се разделят с точка и запетая (;). Редът им няма значение, както и малките и големи букви.

Основните параметри на връзката (Connection String) са следните:

- **Provider** – име на драйвера за достъп до базата.
- **Data Source/Server** – идентификатор на базата. Съдържа името на сървъра или неговия IP адрес.
- **Database/Initial Catalog** – името на базата от данни, която ще се използва в следващите операции (на един сървър може да има няколко бази от данни и трябва да се избере една от тях).
- **User ID/Password** – идентификатор на потребителския account за базата от данни и съответстващата му парола.
- **Integrated Security** – по подразбиране стойността на този параметър е **false**. Ако има стойност "**true**" или "**SSPI**" доставчикът на данни опитва да се свърже към базата с използване на Windows автентикация.
- **Persist Security Info** – ако е **false** (стойността по подразбиране), паролата се премахва от свойството **ConnectionString** веднага след установяване на връзката. Целта е да се намали потенциалната възможност някой да открадне паролата по време на работата на приложението след свързването с базата данни.

За създаване на връзка към SQL Server база от данни задължително в Connection String трябва да се укажат параметрите **Data Source**, **Initial Catalog** и информация за автентикацията – или **User ID/Password**, или **Integrated Security**. Следва пример за Connection String:

```
Server=localhost; Database=Pubs; Integrated Security=true;
Persist Security Info=false;
```

Горният Connection String описва връзка към базата от данни **Pubs** на локалния компютър, като се използва Windows автентикация пред SQL Server.

SqlConnection – пример

Следващият пример илюстрира използването на параметри за връзка (Connection String) и свързване към MS SQL. В началото се задава стойност на константата **CONNECTION_STRING**, която след това се подава като параметър на конструктора на класа **SqlConnection** при създаването на обект от този клас. След отваряне на връзката с метода **Open()** тя може да се използва за изпълнение на заявки към базата данни.

```
const string CONNECTION_STRING =
   "Server=localhost; Database=Northwind; " +
```

```
    "Integrated Security=true; " +
    "Persist Security Info=false";

    // Create the connection
    SqlConnection con = new SqlConnection(CONNECTION_STRING);

    // Open the connection
    con.Open();

    // Use the connection
    using (con)
    {
       // Use the connection here
       // ...
    }
```

Connection Pooling

Създаването на връзка към база от данни е тежка операция, която отнема много време и заема ресурси. Ако много клиенти често се свързват с приложението като отварят връзка към базата, изпълняват някаква операция и затварят връзката, създаването на връзки ще се превърне в тясно място на системата. За решаване на този проблем се използва техниката "Database Connection Pooling". Тя осигурява по-ефективно използване на връзките към базата от данни и така значително подобрява производителността.

Как работи механизмът на Connection Pooling?

Механизмът на "connection pooling" поддържа "пул" от налични връзки към базата. Когато клиент се опита да отвори връзка се използва готова връзка от пула (разбира се, ако там има свободна), вместо да се създаде нова. Ако всички връзки от пула са заети се отваря нова. Ако пулът не съществува, той се създава при първото извикване на метода **`Open()`** на обект за връзка. При затваряне на връзка тя реално не се затваря, а се връща в пула, готова за повторна употреба.

Има някои ограничения относно повторното използване на връзка. Обикновено връзка се преизползва само ако низът за връзка (connection string) съвпада напълно при няколко заявки за отваряне на връзка към един и същ сървър в рамките на едно приложение.

Възможно е да се задават параметри на механизма на "connection pooling" и това става чрез **`ConnectionString`** свойството. Там могат да се зададат например минимален и максимален брой връзки в пула.

SqlClient Data Provider използва пул по подразбиране, т. е. не е нужно програмистът да прави каквото и да е за да използва пул при връзка със MS SQL Server.

Реализация на свързан модел в ADO.NET

Както бе обяснено по-рано, **свързан модел** имаме в случаите, когато данните се пазят на сървъра и клиентът е постоянно свързан към него. При този архитектурен модел, за да изпълни своите задачи клиентското приложение постоянно комуникира със сървъра. Важно е да се знае, че този модел е по-малко скалируем от **несвързания модел**, но за това пък чрез него се постига много по-лесно целта всеки един от клиентите да знае каква е ситуацията с базата от данни в дадения момент, когато е отправил запитване, т. е. да работи с актуални данни.

Кога да използваме свързан модел?

Нека да си представим работата на една средно голяма българска фирма за търговия, около 100-200 души, които се намират в един офис. Те са свързани в единна фирмена мрежа. Сред тях има дилъри, които се занимават с приемане на поръчки по телефона, което на практика е процеса на продажба. Фирмата има също складови работници, които трябва да разберат за уговорените от дилърите сделки и да приготвят стоката за изпращане или предаване на клиента. В такава фирма има и ръководители, които трябва да знаят и следят процесите във фирмата чрез изготвени от софтуерния продукт отчети (наясно сме, че всички прилики с реални фирми са напълно случайни, нали?).

За такава фирма най-приложима е архитектурата на свързвания модел, тъй като скалируемостта, която се постига с него е достатъчна за рамките на един офис, макар и огромен. Предимствата на този модел, че всяка една промяна се отразява и вижда моментално от всички, е всъщност най-важното за такъв тип фирми. Без това те не биха могли да прилагат реално какъвто и да е софтуер.

А какво става ако тази фирма има няколко офиса из страната? Отговорът е най-често в използването на свързан модел за всеки отделен офис и създаването на допълнителен продукт, в който посредством несвързан модел, през определено време, да се събират данните от всички офиси и да се обединяват.

Свързаният и несвързаният модели не се изключват взаимно, а често се прилагат заедно.

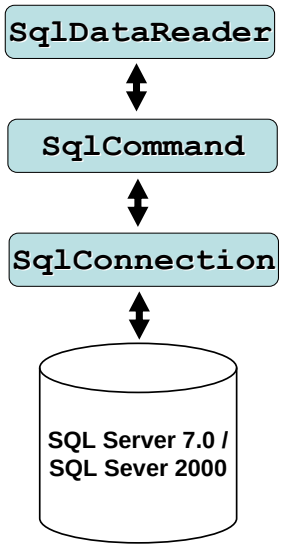
Свързан модел от гледна точка на програмиста

Като идеология и начин на действие в ADO.NET, свързаният модел се имплементира чрез три основни класа:

- клас за **отваряне на връзка (`SqlConnection`)**
- клас за **изпълнение на команда / команди (`SqlCommand`)**

- клас за **обработка на редовете**, получени като резултат от изпълнена заявка, който наричаме **четец (`SqlDataReader`)**

Тези класове имат и свои разновидности и допълнителни имплементации, които, с цел по-голяма яснота и лесната за усвояване на информацията, ще оставим за по-късно. Сега ще наблегнем само на имплементациите, които касаят работа с MS SQL Server. На картинката е показано взаимодействието на споменатите три класа със сървъра и помежду им:

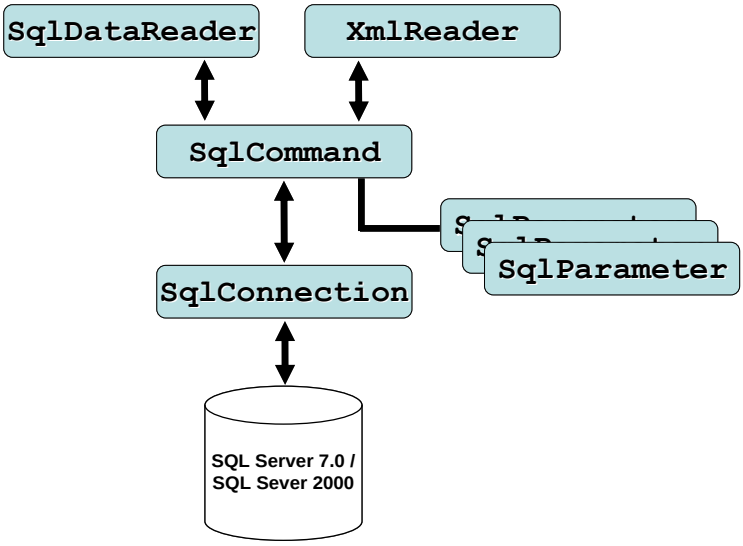


Както е видно от нея, последователността от действия за изпълнение на команда към сървъра са:

1. Отваряне на връзка (`SqlConnection`).
2. Изпълнение на команда / команди (`SqlCommand`).
3. Обработка на редовете, получени като резултат от заявката чрез четец (`SqlDataReader`).
4. Затваряне на четеца.
5. Затваряне на връзката (ако няма да се изпълняват повече команди).

Класовете в свързана среда

На картинката по-долу са показани основните класове, чрез които се имплементира свързан модел в ADO.NET:



Както се вижда, командите, изпълнявани към сървъра, могат да имат и параметри, а получаването на данните от сървъра може да става както с обикновен четец, така и с XML четец, който ги връща в XML формат.

Класът SqlConnection

Както бе обяснено по-рано, чрез инстанции на **`SqlConnection`** се осъществява връзката между MS SQL Server и .NET приложенията. Класът **`SqlConnection`** позволява отваряне и затваряне на връзка и се използва при създаване и изпълнение на команди (**`SqlCommand`**), при работа с адаптери за данни (**`SqlDataAdapter`**) и при започване на транзакция.

Изключително е важно при приключване на работа с базата от данни да затворите връзката към нея, за да не се разхищават ценни ресурси на сървъра. Загубата на ресурси е сериозен проблем и може да доведе до срив на приложението.

При работа с връзка към базата от данни връзката се отваря и затваря или експлицитно, или имплицитно.

Експлицитно отваряне и затваряне на връзка

Експлицитното отваряне на връзка става с явно извикване на метода **`Open()`** на класа **`SqlConnection`**. В този случай се създава връзка към сървъра, или, ако в connection pool има свободни връзки, се взима от пула. Експлицитното затваряне на връзката се извършва с явно извикване на метода **`Close()`** на обекта на връзката. При това връзката се освобождава (и съответно се връща в пула).

Имплицитно отваряне и затваряне на връзка

Имплицитното отваряне и затваряне на връзка става скрито за програмиста и е възможно само при работа с **DataAdapter** – при запълване на **DataSet** обект адаптерът сам отваря връзката и след това я затваря. Поточно, той оставя състоянието на връзката такова, каквото е било – т.е., ако му се подаде отворена връзка той ще извлече данни от базата и ще остави връзката отворена. На това ще се спрем по-подробно в описанието на несвързания модел. При имплицитното отваряне и затваряне на връзки те отново се взимат от пула и връщат в него след приключване на работата с тях.

Използване на метода Dispose()

Методът **Dispose()** се извиква автоматично за обекта на връзката ако използваме конструкцията **using(con)**, където **con** е обектът от клас **SqlConnection** , използван за връзката. В блока **using** връзката се отваря и се използва. След привършване на изпълнението на операциите в блока автоматично се извиква **Dispose()**. Той връща връзката в пула. Този подход е удобен, защото няма опасност да забравим по невнимание да затворим връзката. Методът **Dispose()** на **SqlConnection** вътрешно извиква метода **Close()**.

Събитията на SqlConnection

SqlConnection предлага две събития – **StateChange** и **InfoMessage**. Нека разгледаме за какво служат и как се използват.

Събитието StateChange

Събитието **StateChange** дава информация за това какво се е случило с връзката към базата от данни. То настъпва когато се промени състоянието на връзката. За момента е реализирана функционалност, която предизвиква събитието само в случай на отваряне и затваряне на връзката. Събитието има аргумент от тип **StateChangeEventArgs**, който съдържа информация за предходното и текущото състояние на връзката, съответно в свойствата **OriginalState** и **CurrentState**.

Събитието InfoMessage

Събитието **InfoMessage** дава информация за предупреждения и други съобщения от SQL Server. То има аргумент **SqlInfoMessageEventArg**, в който се съдържа номерът на грешката и текстът на съобщението, както и допълнителна информация. Свойствата на класа **SqlInfoMessageEventArg** са следните:

- **Errors** – представлява колекция от тип **SqlErrorCollection**, която съдържа съобщения за грешки и предупреждения.
- **Message** – съдържа пълния текст на съобщението за грешка.

- **Source** – съдържа името на обекта, генерирал грешката.

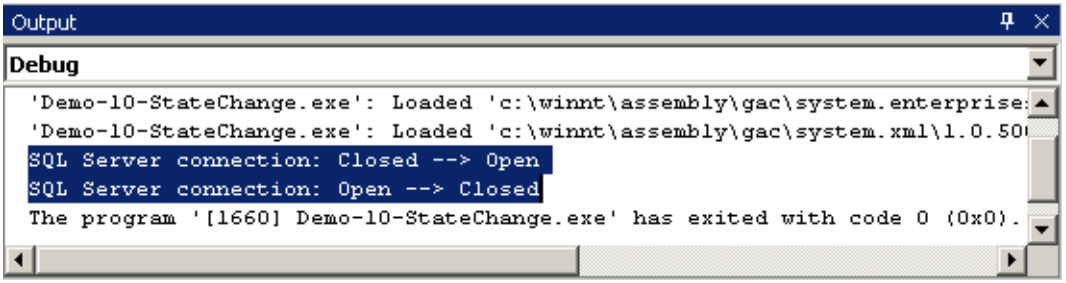
StateChange – пример

Следващият пример илюстрира използването на събитието **StateChange**:

```
private const string CONNECTION_STRING = "Server=.;" +
   " Database=Pubs; Integrated Security=true";

static void Main()
{
   SqlConnection conn = new SqlConnection(CONNECTION_STRING);
   conn.StateChange +=
      new StateChangeEventHandler(ConnStateChange);
   conn.Open();
   conn.Close();
}

private static void ConnStateChange(object sender,
   StateChangeEventArgs e)
{
   Debug.WriteLine("SQL Server connection: " +
      e.OriginalState.ToString() + " --> " +
      e.CurrentState.ToString());
}
```

Описание на примера

Примерът отваря връзка към базата данни и извършва абонамент за събитието **StateChange** на обекта на връзката. На събитието се подава за обработчик методът **ConnStateChange(…)**, който извежда предходното и текущото състояние на връзката. На следващата картинка е показан резултатът от изпълнение на програмата. Редовете, изведени в следствие на възникналото събитие, са маркирани:



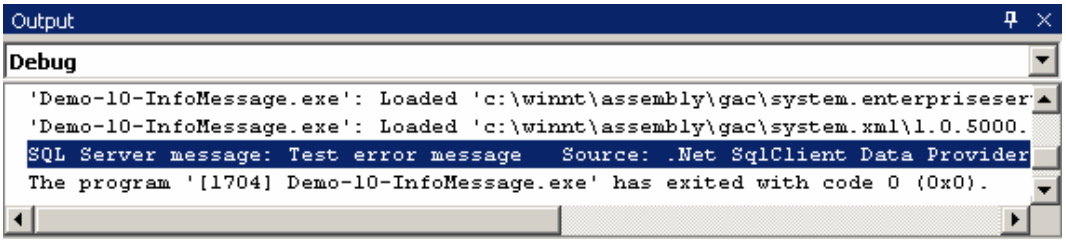
InfoMessage – пример

Следващият пример показва как се използва събитието **InfoMessage**:

```
private const string CONNECTION_STRING = "Server=.; " +
   "Database=Pubs; Integrated Security=true";
```

```
static void Main()
{
   SqlConnection conn = new SqlConnection(CONNECTION_STRING);
   conn.InfoMessage +=
      new SqlInfoMessageEventHandler(ConnInfoMessage);
   conn.Open();
   using(conn)
   {
      SqlCommand cmd = new SqlCommand(
         "RAISERROR('Test error message', 6, 1)", conn);
      cmd.ExecuteNonQuery();
   }
}

private static void ConnInfoMessage(object sender,
   SqlInfoMessageEventArgs e)
{
   Debug.WriteLine("SQL Server message: " +
      e.Message + "; Source: " + e.Source);
}
```

Следващата картинка показва резултата от изпълнението на горния пример. Редът, изведен вследствие на възникналото събитие, е осветен:



Описание на примера

Аналогично на предходния пример, първоначално се създава връзка към SQL Server. След това се извършва абониране за събитието **`InfoMessage`** на обекта на връзката към метода-обработчик **`ConnInfoMessage(…)`**, който извежда съдържанието на полученото от SQL Server съобщение. След това връзката към базата данни се отваря и се изпълнява T-SQL командата:

```
RAISERROR('Test error message', 6, 1)
```

Тази команда предизвиква възникване на грешка със съобщение "**`Test error message`**", с важност **`6`** и информация за състоянието **`1`**. Грешката се получава от ADO.NET Data Provider посредством събитието **`InfoMessage`**.

Класът SqlCommand

Чрез инстанции на **SqlCommand** се изпълнява обръщение към SQL заявка или съхранена процедура на сървъра. Естествено, това става след като бъде осъществено свързването с базата от данни чрез метода **Open()** на инстанцията на **SqlConnection**.

По-важни свойства на SqlCommand

Свойство **Connection** ни позволява да зададем (или получим стойността на вече зададена) връзка, през която ще се изпълнява командата. Макар, че е позволено да има повече от една връзка към базата от данни от едно приложение, това не се препоръчва. Обикновено всички команди се изпълняват през една и съща връзка. Изключение правят сървърните приложения, които по принцип са предназначени да обслужват едновременно много потребители, например уеб услугите. За тях има обикновено по една връзка за всеки обслужван в даден момент потребител.

Свойство **CommandType** (**тип команда**) е изброимо и може да приема като стойност следните три типа команди:

- **CommandType.StoredProcedure**
- **CommandType.TableDirect**
- **CommandType.Text**

От имената им се досещаме, че те указват съответно дали командата ще бъде за изпълнение на съхранена процедура (тип **StoredProcedure**), за директна работа с таблица (тип **TableDirect**) или е стандартен SQL текст (тип **Text**). Последният тип е най-често използваният, като той може да замести предходните два. Чрез него могат да се извикват и съхранени процедури и да се работи с една или повече таблици.

Свойството **CommandText** съдържа име на съхранена процедура, име на таблица или текста на SQL заявка в зависимост от споменатите типове команди.

Свойството **Parameters** (**параметри**) съдържа колекция от параметри към SQL заявка. Задаването на параметри на заявките е много важна функционалност на модерните библиотеки за достъп до релационни бази от данни. Като такава библиотека, SQL Server Data Provider ни предоставя тази възможност. Параметрите ще дискутираме обширно малко по-нататък. Сега ще разгледаме методите, които предлага класът **SqlCommand**.

По-важни методи на SqlCommand

Методът **ExecuteScalar()** на командата връща единична стойност (първата колона от първия ред от резултата). Този метод се ползва само в случаите, когато искаме да получим единична стойност, върната при изпълнение на SQL заявка или съхранена процедура. Върнатата стойност е от тип

System.Object и трябва да се конвертира към съответния очакван тип. Можем да използваме метода например за извикване на съхранена процедура, която добавя нов запис в дадена таблица и връща неговия първичен ключ, генериран от сървъра.

За изпълнение на SQL заявка и извличане на съвкупност от данни, се използва методът **ExecuteReader()** на **SqlCommand**. Той връща обект от класа **SqlDataReader**, който предоставя курсор за навигация по върнатия резултат от изпълнената команда. **ExecuteReader()** се ползва в случаите, когато искаме да получим съвкупност от редове и колони с данни, например при **SELECT** заявки.

Чрез незадължителния параметър **CommandBehavior** се задават настройки на метода, от които се определя дали да се затвори връзката към базата от данни след затваряне на **DataReader** обекта (**CloseConnection**), дали да се върне само един ред (**SingleRow**) или само единична стойност (**SingleResult**) и др. По подразбиране връзката към базата от данни не се затваря, а резултатът от заявката се състои от множество редове и колони. Съществуват и още няколко настройки (**KeyInfo**, **SchemaOnly** и **SequentialAccess**), които се ползват по-рядко.

Методът **ExecuteNonQuery()** се използва при изпълнение на DML заявки за промяна на таблица (**INSERT**, **UPDATE**, **DELETE**). Той връща броя на засегнатите от заявката записи. Типът на резултата е целочислен (**int**).

Методът **ExecuteXmlReader()** се поддържа само в **SqlClient** и обхваща възможност която се поддържа от MS SQL Server – да се връща резултатът в XML формат, като обект от класа **XmlReader**.

Класът SqlDataReader

Чрез инстанции на **SqlDataReader** се осъществява обработката на получените записи след изпълнение на команда (**SqlCommand**). Те са достъпни под формата на курсор, който може да се преглежда в посока само напред (**forward-only**) и да се ползва само за четене (**read-only**). Чрез този клас не могат да се правят промени по данните, нито той може да се обхожда, сортира и т.н.

Целта на му **SqlDataReader** е да се получи резултат от **SELECT** заявка или съхранена процедура (да не забравяме, че и съхранените процедури могат да връщат резултат) и да се обработят данните, които са върнати, без да се променят. За промяна на данните в базата данни се ползват команди (**SqlCommand**).

По-важни методи и свойства на SqlDataReader

Свойство **Item** (индексатор в C#) извлича, стойността на колона по зададено име или индекс. Типът на върнатата стойност зависи от типа на извличаната колона.

Методът **`Read()`** придвижва курсора напред и връща **`false`** ако няма следващ запис.

Методът **`Close()`** затваря курсора. Когато приключим работа с инстанцията на **`SqlDataReader`**, задължително трябва да затворим курсора. В противен случай рискуваме да предизвикаме загуба на ресурси.

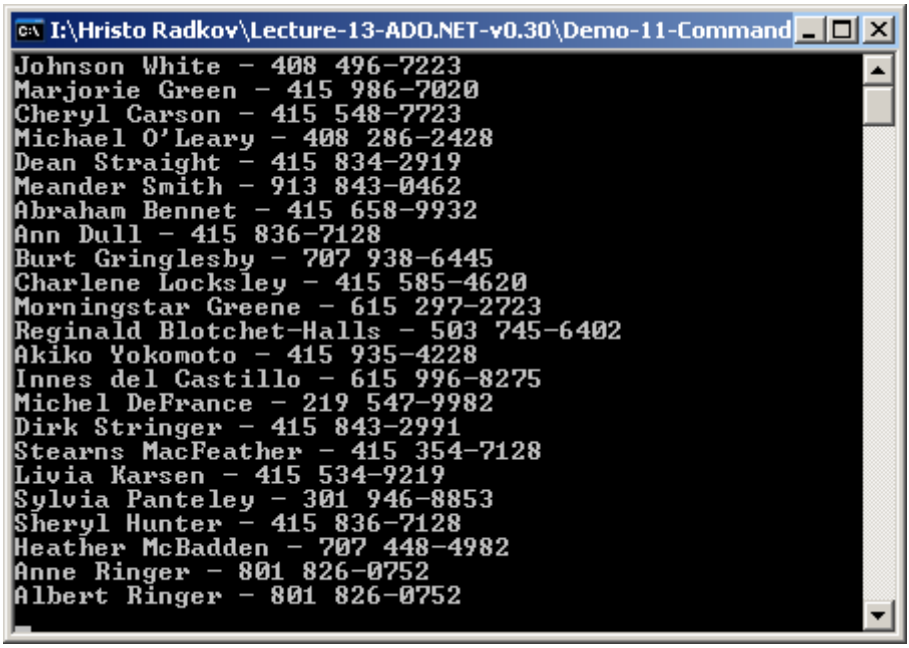
Свързан модел – пример

Следващият пример демонстрира осъществяване на SQL заявка към MS SQL Server и извличане на данни от него при свързан модел на работа:

```
using System;
using System.Data;
using System.Data.SqlClient;

class TestSqlCommand
{
  private const string CONNECTION_STRING = "Server=.; " +
    "Database=pubs; Integrated Security=true";
  private const string COMMAND_SELECT_AUTHORS =
    "SELECT au_fname, au_lname, phone FROM authors";

  static void Main()
  {
    SqlConnection con = new SqlConnection(CONNECTION_STRING);
    con.Open();
    try
    {
      SqlCommand command =
        new SqlCommand(COMMAND_SELECT_AUTHORS, con);
      SqlDataReader reader = command.ExecuteReader();

      using (reader)
      {
        while (reader.Read())
        {
          string firstName = (String) reader["au_fname"];
          string lastName = (String) reader["au_lname"];
          string phone = (String) reader["phone"];
          Console.WriteLine("{0} {1} - {2}",
            firstName, lastName, phone);
        }
      }
    }
    finally
    {
      con.Close();
    }
  }
```

```
}
```

При неговото изпълнение се получава следният резултат:



Описание на примера

В началото към програмата се включват пространствата от имена за работа със свързан модел – **`System.Data`** и **`System.Data.SqlClient`**. В метода **`Main()`** се дефинира и отваря връзка към сървъра посредством **`SqlConnection`** обект с име **`con`**. Използва се базата данни "**`pubs`**", която се доставя като демонстрационен пример заедно с MS SQL Server.

След това се създава инстанция на **`SqlCommand`** с SQL заявка за извличане на данни от таблицата **`authors`** през връзката **`con`**. След изпълнение на командата посредством **`command.ExecuteReader()`** се връща обекта **`reader`** от тип **`SqlDataReader`**.

От този момент нататък вече работим с обекта **`reader`**, който съдържа курсор за обхождане на резултата от заявката. Обхождаме **`reader`** ред по ред докато стигнем до последния. За всеки ред отпечатваме на конзолата данните на авторите в удобен за разглеждане вид. Те се извличат от трите колони с имена съответно **`au_fname`**, **`au_lname`** и **`phone`**.

Накрая задължително затваряме връзката във **`finally`** блока чрез **`con.Close()`** (по точно, препоръчваме на MS SQL Server да я затвори или да я сложи в своя **connection pool**, в зависимост от това как е настроена връзката към сървъра).

Създаване на SqlCommand

Създаването на **`SqlCommand`** обект може да се извърши по три начина**:**

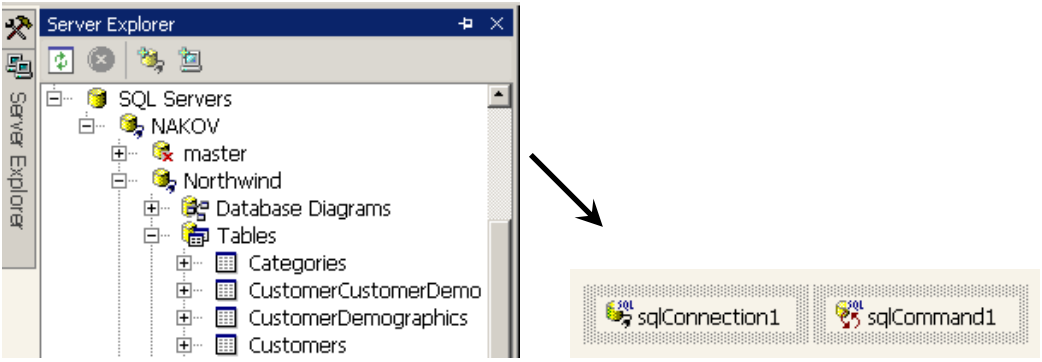
- програмно, както бе направено в примера по-горе;
- по време на дизайн от **Server Explorer** във VS.NET;
- по време на дизайн от **Toolbox** във VS.NET.

Създаване на SqlCommand чрез Server Explorer

Много полезен инструмент за работа със сървъри за бази от данни ни предоставя работната среда на VS.NET. Той се нарича **Server Explorer** (вж. картинката по-долу) и се извиква от меню **View** на VS.NET или чрез клавишна комбинация **[Ctrl + Alt + S]**.

От Server Explorer могат да се разглеждат и дори редактират всякакви обекти от MS SQL Server (таблици, изгледи, съхранени процедури и др.).

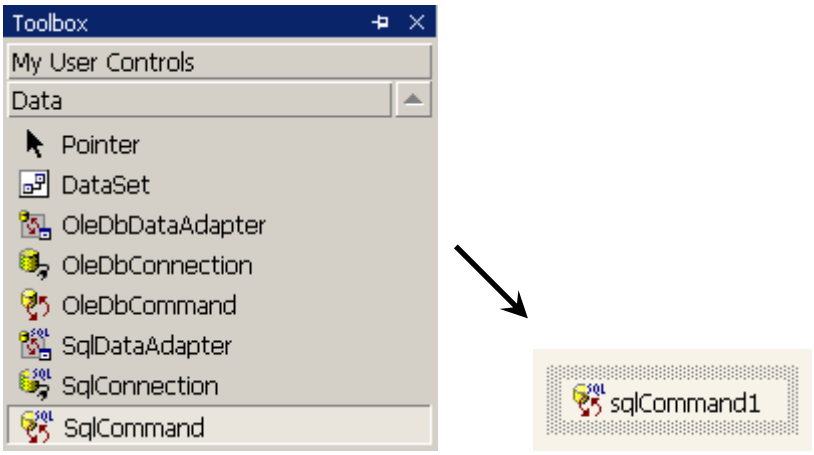
Чрез обикновено влачене и пускане на съхранена процедура в съществуваща форма или друг контейнер за компоненти, средата за разработка VS.NET създава за нас един обект от тип **`SqlConnection`** за връзка към базата от данни и един обект за команда от тип **`SqlCommand`**:



По-късно можем да използваме тези обекти за да извикваме съхранената процедура.

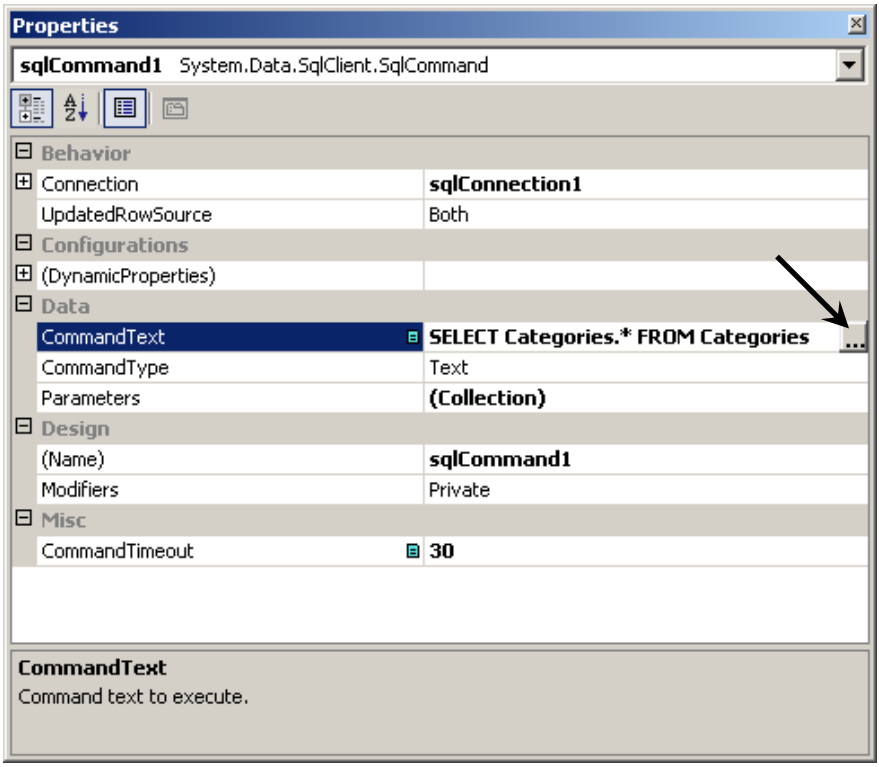
Създаване на SqlCommand чрез Toolbox

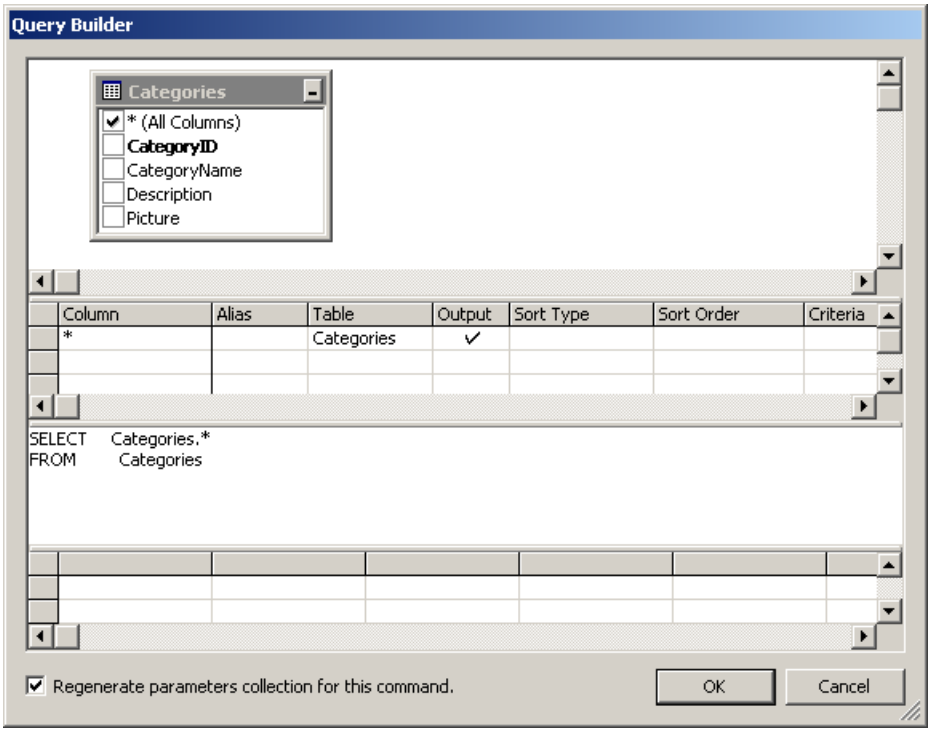
Друг лесен начин да създадем по време на дизайн **`SqlCommand`** обект е чрез палитрата **Toolbox** на VS.NET (на долната картинка). Това става отново чрез влачене и пускане – влачим компонента **`SqlCommand`** върху съществуваща форма или друг контейнер за компоненти:



Тъй като `SqlCommand` е компонент и следователно е наследник на `System.ComponentModel.Component`, той може да се поставя само в т. нар. "контейнери за компоненти", каквито например са Windows формите и уеб формите.

По-късно за настройка на командата се използва редакторът за свойствата на компонентите **Properties** (на долната картинка) и вграденият редактор за команди **Query Builder** на VS.NET, който се появява когато се натисне бутонът  за свойството `CommantText` на командата.





Параметрични SQL заявки

Параметрите са безценно средство при работа със SQL заявки. Чрез тях се решават няколко сериозни проблема, свързани със сигурността на приложението и конвертирането на дати, числа с плаваща запетая и други типове данни.

Параметрите представляват променливи, които се подават към сървъра или се връщат от него. Те могат да се разглеждат и обработват отделно, независимо от заявката. При използването на параметрични заявки първо се изпраща текстът на заявката, а след това за всеки от входните й параметри се задава стойност. След изпълнението й от нея могат да се извличат стойностите на изходните й параметри.

Ако не се използват параметри, всички подадени в заявките данни, трябва да са в текстов формат, защото езикът SQL е текстово базиран. Ако се използват параметрични заявки, обаче, при задаване на стойност на параметър се указва и неговият тип. Така не се налага данните да се преобразуват до текстов формат, което може да създаде много проблеми.

Необходимост от параметрични заявки

Много са причините за използването на параметрични заявки. Нека разгледаме по-важните от тях.

Независимост от формата на данните

SQL заявките са изцяло съставени от текст, например:

```
SELECT * FROM Users WHERE RegistrationDate='1/7/2004'
```

Когато сървърът парсва този текст, датите и десетичните запетаи в него трябва да съответстват на неговите регионални настройки. В противен случай получаваме съобщение от вида "Syntax error converting datetime from character string" или просто некоректна дата.

За да се погрижим това да не се получава, можем да използваме параметри. Другият вариант е да се постараем да подаваме датите и числата с плащаща запетая в "любимия на сървъра" формат, който обаче може да се промени по всяко време без наше знание и тогава ще трябва да пренапишем всички заявки или те просто няма да работят коректно.

Използването на параметри ни дава спокойствието, че сървърът ще обработи данните в тях правилно и ако се наложи ще ги конвертира до съвместим с неговите регионални настройки формат.

Горната заявка можем да направим параметрична по следния начин:

```
SELECT * FROM Users WHERE RegistrationDate = @RegDate
```

В случая задаваме параметър на заявката с име **`@RegDate`**, който може да приема различни стойности, включително обекти от .NET типа **`DateTime`**, при което отпадат проблемите с формата на датите, защото тя не се подава като текст

По-голяма сигурност

Друга важна полза от параметрите е по-добрата защита от атаки от тип **SQL injection**. При тази атака се използва стандартната функция за конкатениране на низове. Тъй като текстът на SQL командите се пази в символен низ, в него (или някъде в него) спокойно могат да се добавят допълнителни инструкции към сървъра за изпълнение на създадена от хакера допълнителна заявка.

За илюстрация на SQL injection атаката ще използваме следния код:

```
string username = "nakov";
string password = "!secret";
string query = "SELECT COUNT(*) FROM Users WHERE UserName = '" +
  username + "' and Password = '" + password + "'";
SqlCommand cmd = new SqlCommand(query, mDbCon);
int rowCount = (int) cmd.ExecuteScalar();
bool authenticated = (rowCount != 0);
```

Примерът има за цел да провери валидността на двойка потребителско име и парола, като ги търси в таблицата **`Users`**. На пръв поглед всичко

изглежда правилно, но това съвсем не е така. В примера се сглобява динамично SQL заявка, която е уязвима на SQL injection атака.

Ако в горния пример за парола се подаде следният низ:

```
' or ''='
```

автентикацията ще бъде успешна, независимо от това какви потребители има регистрирани, защото заявката, която ще се получи:

```
SELECT COUNT(*) FROM Users
WHERE UserName = 'nakov' and Password = '' or ''=''
```

ще връща всички редове от таблицата **Users**.

Проблемът е дори по-сериозен. Помислете само какво ще стане ако за парола бъде подаден низът:

```
'; DROP TABLE Users; SELECT '
```

За избягване на такива ситуации на помощ идват параметричните заявки.

Никога не сглобявайте динамично SQL заявки, ако не сте сигурни, че участващите в тях параметри съдържат единствено и само безопасни за езика SQL символи. Винаги предпочитайте параметрични заявки вместо тях.

Подобрено бързодействие

При използването на параметрични заявки бързодействието може да се подобри, защото заявките се изпращат на сървъра еднократно, а не при всяко изпълнение. При изпращане на параметрична заявка, сървърът първо я анализира и подготвя план за нейното изпълнение (execution plan). След това, при извикването ѝ, сървърът само приема параметрите, които са необходими, и изпълнява вече подготвения план.

При извикване на няколко еднакви заявки, които се различават само по подадените им параметри, техният SQL текст се парсва само веднъж и така се спестява работата на сървъра, свързана с изготвянето на план за изпълнение.

Класът SqlParameter

В .NET Framework работа с параметри при изпълнение на SQL команди към MS SQL Server се извършва от класа **SqlParameter**. Всяка команда **SqlCommand** съдържа колекцията **Parameters** от **SqlParameter** обекти, в която се съдържат всичките ѝ параметри. SQL заявките и съхранените процедури могат да имат входни и изходни параметри.

По важните свойства на класа **SqlParameter** са:

- **ParameterName** – име на параметъра
- **DbType** – тип (**NVarChar**, **Timestamp**, …)
- **Size** – размер на типа (ако има, например ако типа е **NVarChar**, големината може да е 20 символа).
- **Direction** – посока на подаване на параметъра (входен, изходен, двупосочен или резултат от съхранена процедура).

Параметрични заявки – пример

Следващият пример демонстрира осъществяване на SQL заявка с параметри към MS SQL Server. За краткост кодът, който не е съществен за примера, е заменен с многоточие.

```
using System;
using System.Data;
using System.Data.SqlClient;

class SqlParametersTest
{
  private const string CONNECTION_STRING = "Server=.;" +
    " Database=Northwind; Integrated Security=true";

  private SqlConnection mDbCon;

  private void ConnectToDB()
  {
    mDbCon = ...
  }

  private void DisconnectFromDB()
  {
    ...
  }

  private decimal InsertShipper(string aName, string aPhone)
  {
    SqlCommand cmdInsertShipper = new SqlCommand(
      "INSERT INTO Shippers(CompanyName, Phone) " +
      "VALUES (@Name, @Phone)", mDbCon);

    SqlParameter paramName =
      new SqlParameter("@Name", SqlDbType.NVarChar);
    paramName.Value = aName;
    cmdInsertShipper.Parameters.Add(paramName);

    SqlParameter paramPhone =
      new SqlParameter("@Phone", SqlDbType.NVarChar);
    paramPhone.Value = aPhone;
```

```
      cmdInsertShipper.Parameters.Add(paramPhone);

      cmdInsertShipper.ExecuteNonQuery();

      SqlCommand cmdSelectIdentity =
        new SqlCommand("SELECT @@Identity", mDbCon);

      decimal insertedRecordId =
        (decimal) cmdSelectIdentity.ExecuteScalar();
      return insertedRecordId;
    }

    static void Main()
    {
      SqlParametersTest test = new SqlParametersTest();
      try
      {
        test.ConnectToDB();
        ...

        // Insert new shipper in the "Shippers" table
        decimal newShipperId =
          test.InsertShipper("Тест", "123-456-789");
        Console.WriteLine("Inserted new shipper. " +
          "ShipperID = {0}", newShipperId);
      }
      finally
      {
        test.DisconnectFromDB();
      }
    }
}
```

Описание на примера

В примера е дефиниран клас **`SqlParametersTest`** с метод **`Main()`**, който при извикване създава инстанция на класа и извиква метода за свързване с базата от данни. Забележете, че при отваряне на връзката веднага се поставя **`try-finally`** блок, като във **`finally`** частта се затваря връзката. Това трябва да се приема от съвременните програмисти като задължителна (а не просто препоръчителна) практика.

Сега да се върнем на примера. След като е осъществена връзка към базата данни, се изпълнява заявка за добавяне на нов превозвач в таблица **`Shippers`**. Това става в метода **`InsertShipper()`**, на който се подават данните за новия превозвач. Ще разгледаме по-обстойно този метод, тъй като в него се изпълнява SQL заявка с параметри.

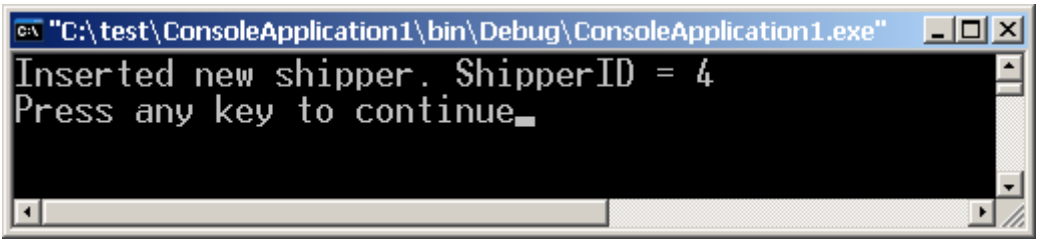
Първо се дефинира командата за добавяне на запис в таблица **`Shippers`**. Тя съдържа следната заявка: **`"INSERT INTO Shippers(CompanyName,`**

Phone) VALUES (@Name, @Phone)". Забележете, че вместо стойности в частта **VALUES** са дадени идентификатори, предшествани от знака '@'. Този знак указва на MS SQL Server, че следва име на параметър, а не стойност.

След като вече сме създали командата, но преди да я изпълним, създаваме двата параметъра **@Name** и **@Phone** като им указваме име и тип (за случая това е достатъчно), задаваме им стойност и стартираме изпълнението на заявката чрез **cmdInsertShipper.ExecuteNonQuery()**.

Така резултатът от заявката е същият, като от заявката "**INSERT INTO Shippers(CompanyName, Phone) VALUES ('Тест', '123-456-789')**". Разликата е, че ако трябва да конструираме тази заявка чрез конкатенация на низове, и ако данните за новия превозвач се вземат от потребителски интерфейс, в който потребителите могат да въвеждат свободно текст (както става най-често), рискуваме да получим неочакван текст на заявката (спомняте си примерите за **SQL injection**, нали?).

Резултатът от изпълнението на примера е показан на картинката:



Както се вижда от резултата, след изпълнението на заявката за добавяне на запис се извлича и неговият пореден номер (стойността на първичния ключ **ShipperID** от таблицата **Shippers**), който MS SQL Server услужливо генерира вместо нас. Нека разгледаме как точно става това.

Първичен ключ с пореден номер – извличане

Извличането на автоматично-генериран първичен ключ е специфично за всеки сървър за бази от данни. При **MS SQL Server** и **MS Access** се използват съответно **Identity** и **AutoNumber** колони, при **Oracle** това се прави чрез **Sequence**, при **InterBase/Firebird** чрез **Generator** и т.н.

При MS SQL Server и MS Access стойността на автоматично генериран първичен ключ се извлича със следната заявка:

```
SELECT @@Identity
```

Тази заявка се изпълнява непосредствено след изпълнението на **INSERT** заявка за добавяне на нов запис в таблица. Служебната променлива **@@Identity** в MS SQL Server и MS Access съдържа последния автоматично генериран първичен ключ при **INSERT** заявка, изпълнена през текущата връзка с базата данни. Това е стойността на колоната от добавения запис, която е маркирана като **Identity** и за която MS SQL Server се грижи да

получава автоматично нова уникална стойност при всяко добавяне на запис. Ако такава колона няма, променливата **@@Identity** не съдържа смислена стойност.

В предходния пример веднага след изпълнението на заявката за добавяне на превозвач се извиква заявката за извличане на поредния номер на превозвача (първичния ключ). Този номер съответства на **ShipperID** колоната, която е **Identity** за таблицата **Shippers**. Самото извличане става по следния начин:

```
SqlCommand cmdSelectIdentity =
  new SqlCommand("SELECT @@Identity", mDbCon);

decimal insertedRecordId =
  (decimal) cmdSelectIdentity.ExecuteScalar();
```

Стойността на първичния ключ е от тип **decimal**, защото променливата **@@Identity** в SQL Server е от тип **numeric**, а той съответства на типа **System.Decimal** в .NET Framework.

Използване на транзакции

Транзакциите са друга много важна възможност на модерните сървъри за управление на бази от данни. Чрез тях се извършва изолация и синхронизация на достъпа до данните от различни работни места, както и едновременното потвърждение (**commit**) или отказване (**rollback**) на поредица от заявки.

В настоящата тема ще разгледаме техническите средства за използване на транзакции в ADO.NET и няма да се спираме в дълбочина на проблемите на консистентността на данните и конкурентния достъп. Очаква се читателят да знае, че когато за извършването на една бизнес операция се изисква добавяне или промяна на данни в повече от една таблица, най-често е необходимо отделните заявки, които операцията включва, да се изпълняват в транзакция.

Работа с транзакции в SQL Server

Както вече знаем, работата с транзакции в SQL Server се управлява чрез SQL командите:

- **BEGIN TRANSACTION** – започва транзакция;
- **COMMIT TRANSACTION** – потвърждава текущата транзакция;
- **ROLLBACK TRANSACTION** – отказва текущата транзакция;
- **SET TRANSACTION ISOLATION LEVEL [level]** – задава ниво на изолация за текущата транзакция.

Понякога работата с транзакциите трябва да се извърши на по-високо ниво – не в самия сървър на базата данни, а в бизнес логиката на приложението. В такива случаи се използват транзакциите на ADO.NET.

Работа с транзакции в ADO.NET

Както може да се очаква, ADO.NET предлага множество функции за лесна програмна работа с транзакции. Основните възможности, които задължително се налага да ползваме са:

- започване на транзакция:

```
SqlTransaction trans = dbConnection.BeginTransaction();
```

- въвличане на команда в дадена транзакция:

```
command.Transaction = trans;
```

- потвърждаване на транзакция:

```
trans.Commit();
```

- анулиране на транзакция:

```
trans.Rollback();
```

Преди да пристъпим към демонстрация на работата с транзакции трябва да спомена и възможностите, които предоставя ADO.NET за задаване на едно важно свойство на транзакциите – ниво на изолация.

Нивото на изолация се дефинира с изброения тип **`IsolationLevel`** на класа **`SqlTransaction`**. Типовете **`IsolationLevel`**, както можем да очакваме, са почти същите като в SQL Server:

- **`ReadUncommited`** – позволява текущата транзакция да чете непотвърдени данни, добавени или променени от други, паралелно изпълняващи се, транзакции. Не осигурява повторяемост при последователно четене на едни и същи данни и не предпазва от "фантомни записи" (записи, които се появяват в дадена таблица по време на транзакцията в резултат работата на друга, паралелна транзакция).
- **`ReadCommited`** – позволява текущата транзакция да вижда само вече потвърдени данни. Промените от другите паралелно изпълняващи се транзакции, които не са приключили успешно, са невидими за текущата. Не се осигурява повторяемост при последователно четене на едни и същи данни. Не се осигурява защита от "фантомни записи".
- **`RepeatableRead`** – заключва всички данни, четени от текущата транзакция докато тя не приключи, с което се осигурява повторяемост на при четенето. Не предпазва от "фантомни записи".

- **Serializable** – заключва всички таблици, с които работи текущата транзакция по такъв начин, че останалите, паралелно изпълняващи се, транзакции да не могат да променят и добавят данни в тях докато текущата не завърши. Осигурява повторяемост при четенето и защита от "фантомни записи"
- **Chaos** – липса на изолация. Промените от транзакциите с по-висока изолация са с предимство.
- **Unspecified** – друго, нестандартно ниво на изолация.

При започване на транзакция може да се укаже нейното ниво на изолация по следния начин:

```
SqlTransaction trans =
  dbConnection.BeginTransaction(IsolationLevel.Serializable);
```

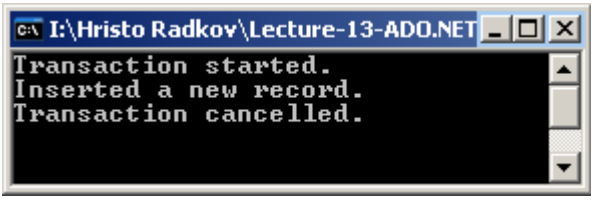
Транзакции – пример

Следващият пример демонстрира създаването на транзакция в ADO.NET и изпълнението на поредица от заявки в нея:

```
using System;
using System.Data;
using System.Data.SqlClient;

class TestTransactions
{
  private const string CONNECTION_STRING = "Server=.;" +
    " Database=Northwind; Integrated Security=true";

  static void Main()
  {
    SqlConnection dbCon = new SqlConnection(CONNECTION_STRING);
    dbCon.Open();
    try
    {
      SqlTransaction trans =
        dbCon.BeginTransaction(IsolationLevel.ReadCommitted);
      Console.WriteLine("Transaction started.");

      SqlCommand cmd = dbCon.CreateCommand();
      cmd.Transaction = trans;
      try
      {
        cmd.CommandText =
          "INSERT INTO Shippers(CompanyName, Phone) " +
          "VALUES ('New record', '111-111-1111')";
        cmd.ExecuteNonQuery();
        Console.WriteLine("Inserted a new record.");
```

```
            // Този insert ще доведе до изключение понеже
            // поле CompanyName не може да приема стойност null
            cmd.CommandText =
               "INSERT INTO Shippers(CompanyName, Phone) " +
               "VALUES (null, '123-456-7890')";
            cmd.ExecuteNonQuery();
            Console.WriteLine("Inserted a new record.");

            trans.Commit();
            Console.WriteLine("Transaction comitted.");
         }
         catch (SqlException)
         {
            trans.Rollback();
            Console.WriteLine("Transaction cancelled.");
         }
      }
      finally
      {
         dbCon.Close();
      }
   }
}
```

Резултатът от изпълнението на примера е следният:



Описание на примера

В примера се отваря връзка към базата данни **Northwind** от локалния SQL Server. След като връзката е осъществена успешно, се създава транзакция с име **trans**. След това се създава нова команда **cmd** и чрез реда **cmd.Transaction = trans** се указва тя да се изпълнява през транзакцията **trans**. След това през командата **cmd** се изпълняват два пъти последователно различни SQL заявки. Забележете, че през една инстанция на **SqlComand** могат да се изпълняват множество заявки с промяна единствено на SQL текста в полето **CommandText**. И двете заявки имат за цел добавяне на нов превозвач в таблица **Shippers**.

За да е по-интересно, при втората заявка **INSERT INTO Shippers(CompanyName, Phone) VALUES (null, '123-456-7890')** ще опитаме да нарушим консистентността на данните, като в полето **CompanyName**, което не допуска празни стойности, зададем стойност **null**. Естествено сървъ-

рът няма да допусне това да се случи и като резултат ще получим изключение, което е редно да обработим в **try-catch** блок.

Нормално е, тъй като желаем да изпълним и двете SQL команди заедно (нали затова са в транзакция), или и двете да минат успешно, или да бъде отказано изпълнението и на двете. Затова след изпълнението на двете команди се поставя кодът **trans.Commit()**, чрез който се потвърждават промените в транзакцията, освен ако преди това не се е появила грешка. В **catch** частта на **try-catch** блока се обработва евентуалната грешка, която може да възникне, и се извиква методът **trans.Rollback()**, чрез който се отказва транзакцията. Така или се изпълняват и двете заявки и се потвърждава транзакцията, или някоя от заявките не успява и транзакцията се анулира.

От концептуална гледна точка за всички сървъри за бази от данни транзакциите се използват за успешното изпълнение или отказване на поредица от SQL заявки. Затова задължително тази поредица от заявки се изпълнява в try-catch блок, като последният ред в try частта трябва да съдържа код за потвърждаване на транзакцията, а в catch частта трябва да има код за отказването ѝ при евентуално настъпване на изключение.

Връзка с други бази от данни

Както споменахме по-рано в тази глава, освен средства за работа с MS SQL Server, ADO.NET поддържа връзка и с други сървъри за управление на бази от данни. Това е възможно благодарение на доставчиците на данни, които вече дискутирахме.

Освен към връзка със сървъра на Microsoft, можем да се възползваме от доставчици на данни за работа и с **Oracle** и **OLE DB**, както и такива за свързване посредством стандарта **ODBC** (Open Database Connectivity). Допълнително от Интернет могат да бъдат изтеглени пакети за работа с най-разпространените RDBMS сървъри (например MySQL, PostgreSQL, Interbase, Firebird и др.).

Като идеология всички класове за достъп до данни са аналогични на тези, които разгледахме за работа с Microsoft SQL Server (**SqlConnection**, **SqlCommand**, **SqlParameter**, **SqlDataReader**, **SqlTransaction** и т.н.).

Например за работа с Oracle се използват класовете: **OracleConnection**, **OracleCommand**, **OracleParameter**, **OracleDataReader**, **OracleTransaction** и т.н. Всички тези класове много си приличат, защото имплементират интерфейсите на ADO.NET за достъп до данни, за които вече стана дума:

- **IDbConnection** – за връзка с базата данни;
- **IDbTransaction** – за използване на транзакции;

- **`IDbCommand`** и **`IDataParameter`** – за изпълнение на команди към сървъра и за параметри към командите;
- **`IDataReader`** – за четене на данни;
- **`IDbDataAdapter`** – адаптер за данни. Ако този интерфейс ви се струва непознат в сравнение с останалите, то това е защото не се използва при свързания модел. Той, обаче, играе важна роля при несвързания модел, който ще разгледаме по-нататък.

OLE DB Data Provider

Както може да се очаква, класовете за достъп до данни през интерфейса **OLE DB** са следните:

- **`OleDbConnection`** – осъществява връзка с OLE DB източник на данни.
- **`OleDbCommand`** – изпълнява SQL команди върху OLE DB връзка към база данни.
- **`OleDbParameter`** – представлява параметър на команда.
- **`OleDbDataReader`** – служи за извличане на данни от команда, изпълнена през OLE DB.
- **`OleDbDataAdapter`** – обменя данни между **`DataSet`** обекти и OLE DB източник на данни. Използва се при несвързания модел на работа.

Стандартът OLE DB

Стандартът **OLE DB** е разработен с основната цел да предостави възможност за работа с бази от данни в Windows. Като такъв той е прилаган и в обектния модел на Microsoft Access.

Ще разгледаме накратко архитектурата на Microsoft Access и Microsoft JET Engine.

Microsoft Access

За разлика от MS SQL Server, Oracle и останалите RDBMS сървъри на пазара, Access не предоставя собствена услуга (service), която да слуша на определен порт (например MS SQL Server слуша на портове 1433 и 1434) за свързване от клиентски приложения към базата от данни. Вместо това той предлага файлов достъп до базата данни (**`.mdb`** или по-рядко **`.mde`** файл), като обработката на информацията във файла се извършва от така наречения Microsoft JET Engine.

Microsoft JET Engine

От архитектурна гледна точка **JET Engine** представлява OLE сървър за работа с бази от данни, който обаче работи на всеки един компютър (клиент), желаещ да се свързва към базата. Така базата се обработва

локално на клиентския компютър, като достъпът до файла с данни може да става по мрежата (чрез обикновен споделен файл).

Този подход, разбира се, увеличава мрежовия трафик и създава предпоставки за по-нестабилна работа и трудности при синхронизацията на достъпа. По тази причина са малко системите, обслужващи конкурентно множество потребители, които използват Microsoft Access файлове за база от данни.

Като изключим този факт, версия 4.0 на JET Engine на Microsoft Access предлага множество възможности за работа с данни (таблици, изгледи, SQL заявки, транзакции и т.н.), близки до тези на нормалните сървъри, но на много по-ниска цена.

Всъщност самият JET Engine е напълно безплатен. Ако сме склонни да се откажем от Microsoft Access (който е част от платения пакет Microsoft Office) като визуална обектно-ориентирана среда за работа с него, можем да инсталираме безплатната библиотека **MDAC** (**Microsoft Data Access Components**), която съдържа и JET Engine. Операционните системи Microsoft Windows 2000/XP/2003 съдържат MDAC, вграден като неразделна част от тях.

OLE DB технологията

OLE DB технологията се използва не само за достъп до MS Access, но и за достъп до всякакви бази от данни, за които има OLE DB драйвери (например Oracle, Interbase, MySQL и др.). На практика OLE DB стандартът малко по малко взе надмощие над ODBC стандарта поради по-съвременния си подход и се утвърди като основен начин за достъп до релационни бази от данни в Windows.

Връзка към OLE DB

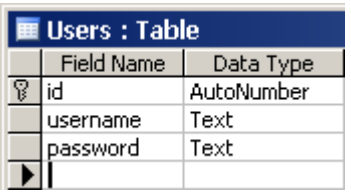
И така, връзката към OLE DB в ADO.NET се осъществява чрез **`OleDbXXX`** класовете, като за **`OleDbConnection`** класа се използва connection string, подобен на следния:

```
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\some_file.mdb;
Persist Security Info=False
```

Разликите между достъпа до OLE DB и SQL Server база от данни не са големи, както ще видим в примера. Ако не са използвани специфични за SQL Server команди и класове, кодът писан за SQL Server може лесно да се преправи да работи и с OLE DB база от данни.

Връзка с OLE DB – пример

За целта на примера сме създали с MS Access база от данни, която се намира във файл **`C:\Library.mdb`**. В нея сме направили таблица **`Users`** със структура, показана на картинката:



За да се свържем с тази `.mdb` база данни използваме **"Microsoft Jet 4.0 Provider"**. Той осигурява връзка от ADO.NET към JET Engine през OLE DB. Нека да разгледаме един пример за достъп до базата данни **`Library.mdb`**:

```
using System;
using System.Data.OleDb;

class OleDbConnectionDemo
{
  const string MS_ACCESSS_GENERIC_CONNECTION_STRING =
    @"Provider=Microsoft.Jet.OLEDB.4.0;Data Source={0};" +
    @"Persist Security Info=False";

  static void Main()
  {
    string connectionString = string.Format(
      MS_ACCESSS_GENERIC_CONNECTION_STRING,
      @"C:\Library.mdb");

    OleDbConnection dbConn = new
      OleDbConnection(connectionString);

    // Open connection
    dbConn.Open();
    using (dbConn)
    {
      OleDbCommand cmd = new OleDbCommand(
        "INSERT INTO Users ([username], [password]) " +
        "VALUES (@user, @pass)", dbConn);
      cmd.Parameters.Add("@user", OleDbType.VarChar).Value =
        "new user name";
      cmd.Parameters.Add("@pass", OleDbType.VarChar).Value =
        "secret password";

      // Execute the INSERT command
      try
      {
        int rowsAffected = cmd.ExecuteNonQuery();
        if (rowsAffected > 0)
        {
          OleDbCommand cmdSelectIdentity =
            new OleDbCommand("SELECT @@Identity", dbConn);
          int insertedRowId =
            (int) cmdSelectIdentity.ExecuteScalar();
```

```
            Console.WriteLine("Operation was successfull. " +
              "Inserted row id = {0}.", insertedRowId);
          }
        }
        catch (OleDbException)
        {
          Console.WriteLine("SQL Error occured!");
        }
      }
    }
}
```

Описание на примера

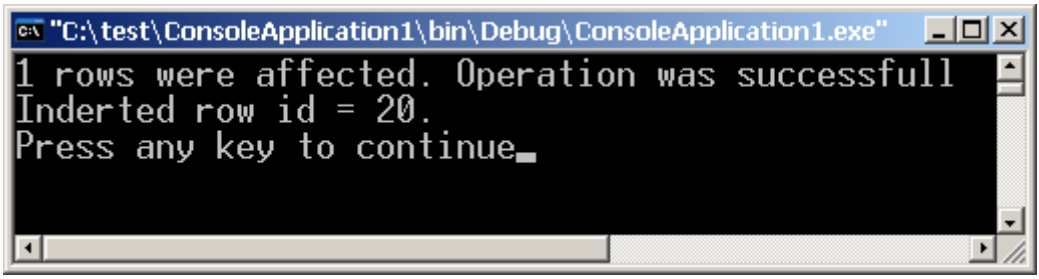
В примера се осъществява връзка към MS Access базата данни посредством класа **`OleDbConnection`**, на който се подава подходящ connection string. Низът за връзка е шаблонен и съдържа форматиращ низ. В него посредством метода **`String.Format(…)`** се вмъква името на файла с базата данни (**`.mdb`** файла).

След това се създава нова команда с име **`cmd`**, инстанция на класа **`OleDbCommand`** и чрез нея се изпълнява SQL заявка за добавяне на запис в таблицата **`Users`**, като се използват параметри, на които са зададени съответните типове и стойности.

За щастие OLE DB доставчикът на данни и MS Access поддържат параметрични заявки и това значително улеснява работата ни.

След изпълняване на познатия ни от предишните примери код **`cmd.ExecuteNonQuery()`**, се прави проверка колко записа са добавени към таблицата. В случай, че всичко е наред (**`INSERT`** клаузата е минала успешно), се визуализира поредният номер на добавения в таблицата запис. Той се извлича от системната променлива **`@@Identity`**, както това се правеше при MS SQL Server. При настъпване на проблем с някоя от операциите с базата данни възниква и се обработва **`OleDbException`**.

Резултатът от изпълнението на примера е показан на картинката по-долу:



Правилна работа с дати

Когато дискутирахме параметрите по-рано в тази глава, споменахме колко важни са те за правилното предаване на определени типове данни към сървъра, например дати. Именно датите, заради различните регионални настройки, които могат да се очакват на всеки отделен компютър, са един много специален тип данни, с които трябва да се работи внимателно.

За всеки програмист е ясно, че датите могат да бъдат предавани като низ, но както обяснихме преди, това би довело до некоректната работа на нашето приложение на компютър, който има други регионални настройки.

И до ден днешен много програмисти правят грешката да сглобяват (конкатенират) низове при SQL заявки и да предават датите като низ. В резултат на това техните приложения задължават потребителя да има регионални настройки като тези на компютъра за разработка.

Виждали сме случаи когато две или повече "добре написани програми" изискват различни регионални настройки и не могат да се ползват едновременно на един компютър. За да не копирате тъжния опит на тези наши колеги, ви препоръчваме да съблюдавате следните правила:

- Използвайте за датите вградените типове на базата данни, с която работите, а не символен низ (string). Някои сървъри за бази от данни поддържат дори повече от един тип за дати. Например в MS SQL Server 2000 има тип **`datetime`** (8 байта) и **`smalldatetime`** (4 байта), като разликата е в точността.
- При работа с данни от тип дата, използвайте символен низ само за визуализация към потребителя. Не използвайте символни низове когато предавате данни от тип дата от един метод към друг.
- Използвайте **`System.DateTime`** структурата за работа с дати в .NET Framework. Това е типът данни, към който в повечето случаи трябва да преобразувате дати, извлечени от базата данни.
- Използвайте параметрични заявки за предаване на дати към базата данни. Никога не подавайте към базата данни дата като символен низ. Подавайте датите като параметър, който е от тип дата.
- При нужда от конвертиране предавайте **`IFormatProvider`** за дефиниране на правилата за конвертиране. Ако такава нужда възникне, тя би трябвало да е свързана с потребителския интерфейс, при въвеждане или извеждане на дата. В останалите случаи можем да ползваме параметри в SQL заявките.
- При необходимост използвайте неутрални културни настройки (**`CultureInfo.InvariantCulture`**). Това ще ви спести някои проблеми с локализацията в случаите, в които искате просто да изведете или прочетете дата от формат, който не е зависим от езика.

Работа с дати – пример

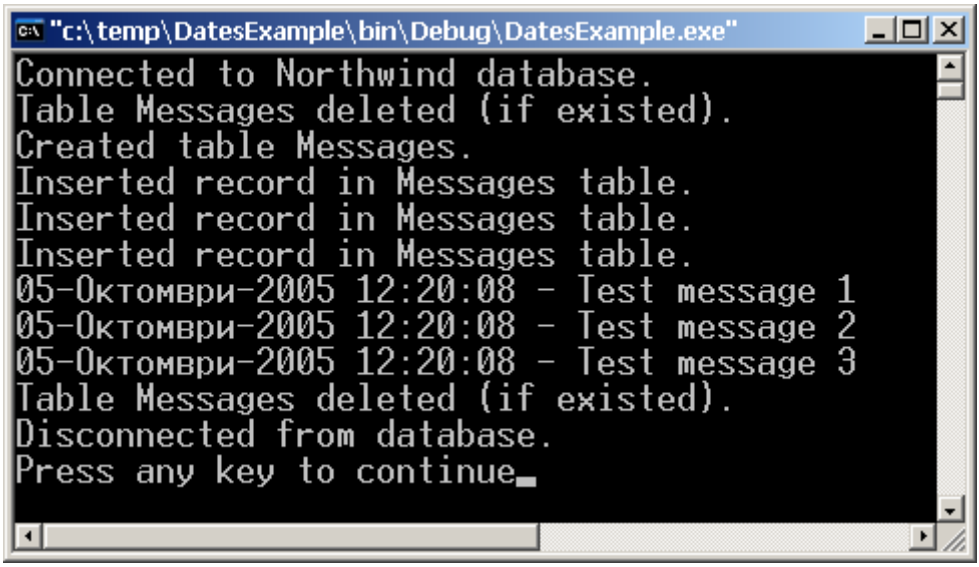
Следващият пример демонстрира правилния начин за работа с дати:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Globalization;

class DatesDemo
{
   const string CONNECTION_STRING = "Server=.; " +
      "database=Northwind; Integrated Security=SSPI";
   private static SqlConnection mDbCon;

   static void Main()
   {
      DatesDemo demo = new DatesDemo();
      demo.ConnectToSqlServer();
      demo.DropMessagesTable();
      demo.CreateMessagesTable();
      demo.AddMessage("Test message 1", DateTime.Now);
      demo.AddMessage("Test message 2", DateTime.Now);
      demo.AddMessage("Test message 3", DateTime.Now);
      CultureInfo cultureBulgaria = new CultureInfo("bg-BG");
      demo.DisplayAllMessages(cultureBulgaria,
         "dd-MMM-yyyy HH:mm:ss");
      demo.DropMessagesTable();
      demo.DisconnectFromSqlServer();
   }

   public void ConnectToSqlServer()
   {
      mDbCon = new SqlConnection(CONNECTION_STRING);
      mDbCon.Open();
      Console.WriteLine("Connected to Northwind database.");
   }

   public void CreateMessagesTable()
   {
      SqlCommand cmdCreateMsgTable = new SqlCommand(
         @"CREATE TABLE Messages
         (
            MsgId int identity not null primary key,
            MsgText nvarchar(1000),
            MsgDate datetime
         )",
         mDbCon
         );
      cmdCreateMsgTable.ExecuteNonQuery();
      Console.WriteLine("Created table Messages.");
```

```
}

public void AddMessage(string aMsgText, DateTime aMsgDate)
{
   SqlCommand cmdInsertMsg = new SqlCommand(
      "INSERT INTO Messages(MsgText, MsgDate) " +
      "VALUES (@MsgText, @MsgDate)", mDbCon);

   SqlParameter paramMsgText =
      new SqlParameter("@MsgText", SqlDbType.NVarChar);
   paramMsgText.Value = aMsgText;
   cmdInsertMsg.Parameters.Add(paramMsgText);

   SqlParameter paramMsgDate =
      new SqlParameter("@MsgDate", SqlDbType.DateTime);
   paramMsgDate.Value = aMsgDate;
   cmdInsertMsg.Parameters.Add(paramMsgDate);

   cmdInsertMsg.ExecuteNonQuery();

   Console.WriteLine("Inserted record in Messages table.");
}

public void DisplayAllMessages(CultureInfo aCultureInfo,
   string aFormat)
{
   SqlCommand cmdSelectMsgs = new SqlCommand(
      "SELECT MsgText, MsgDate FROM Messages", mDbCon);
   SqlDataReader reader = cmdSelectMsgs.ExecuteReader();
   using (reader)
   {
      while (reader.Read())
      {
         string msgText = (string) reader["MsgText"];
         DateTime msgDate = (DateTime) reader["MsgDate"];
         string msgDateFormatted = msgDate.ToString(aFormat,
            aCultureInfo);
         Console.WriteLine("{0} - {1}", msgDateFormatted,
            msgText);
      }
   }
}

public void DropMessagesTable()
{
   SqlCommand cmdCreateMsgTable = new SqlCommand(
      @"IF OBJECT_ID('Messages') IS NOT NULL
         DROP TABLE Messages",
      mDbCon);
   cmdCreateMsgTable.ExecuteNonQuery();
```

```
      Console.WriteLine("Table Messages deleted (if existed).");
   }

   public void DisconnectFromSqlServer()
   {
      mDbCon.Close();
      Console.WriteLine("Disconnected from database.");
   }
}
```

При изпълнението на горната програма се получава следният резултат:



Описание на примера

В примера се отваря връзка към базата данни **Northwind** от локалния SQL Server и в нея се създава нова таблица с име **Messages**, която съдържа текстова колона и колона от тип дата (**datetime**). Таблицата се използва само за демонстрацията и накрая бива унищожена.

След това за да се илюстрира работата с дати в таблицата се добавят няколко записа чрез метода **AddMessage(…)**. Забележете, че той приема датата като параметър от тип **System.DateTime**, а не като символен низ и в параметричната SQL команда, която е използвана, съответният параметър е от тип **SqlDbType.DateTime**.

За да се демонстрира извеждането на дати от базата данни в метода **DisplayAllMessages(…)** се показва на екрана съдържанието на таблицата **Messages**. Обърнете внимание, че методът приема като параметър култура и формат за извежданата дата и ги използва при отпечатването на датите. Добрата практика изисква винаги, когато преобразувате дати към символен низ (например когато ги отпечатвате), да ги форматирате с една

и съща култура и с един и същ формат. Само така можете да си гарантирате, че вашето приложение ще работи правилно независимо от системните езикови и регионални настройки.

Преди завършване на работата на приложението се унищожава временната таблица **Messages**, използвана само за целите на демонстрационния пример.

Работа с картинки в база от данни

В базите от данни могат да се съхраняват различни типове данни. Графичните изображения не правят изключение. За съхраняването на графични обекти в база от данни обикновено се използват бинарни полета. Различните бази използват различни типове данни за съхраняване на бинарни данни.

Двоични данни в MS SQL Server

В MS SQL Server се използва типът данни **image**. Той е един от трите типа за съхраняване на двоични данни (останалите са **binary** и **varbinary**). Препоръчва се **image** да се използва, ако данните са с обем повече от 8 KB. Иначе може да се използва **varbinary**. Типът **image** може да съхранява всякакви бинарни данни: Word документи, Excel таблици, **jpeg**, **gif**, **png** файлове и др.

Двоични данни в Oracle

Oracle използва типа **blob** за съхраняване на бинарни данни и в частност на графични изображения. В Oracle 9i в поле от този тип могат да се съхраняват до 4 GB данни, например картинки, Word документи, филми и др. В базата данни Oracle има и тип **bfile**, който също се използва за съхранение на двоични данни до 4 GB, като разликата е, че при него данните се съхраняват извън базата, във външен файл, а в базата се пази само името на този файл.

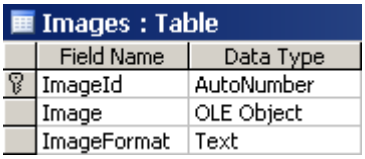
Двоични данни в MS Access

В MS Access за съхранение на бинарни данни и графични изображения се използва типът **OLE Object**.

Съхранение на графични обекти – пример

Следващият пример показва как се записват и четат графични обекти от база от данни.

За съхранението на картинките се използва MS Access база, в която е дефинирана следната таблица:



Images : Table

	Field Name	Data Type
🔑	ImageId	AutoNumber
	Image	OLE Object
	ImageFormat	Text

Ето и сорс кодът на примера:

```
using System;
using System.Collections;
using System.Data.OleDb;
using System.IO;

class ImagesInDBDemo
{
  private const string DB_CONNECTION_STRING =
    @"Provider=Microsoft.Jet.OLEDB.4.0;" +
    @"Data Source=..\..\Images.mdb";

  private const string SOURCE_IMAGE_FILE_NAME =
    @"..\..\logo.gif";
  private const string DEST_IMAGE_FILE_NAME =
    @"..\..\logo-from-db.gif";

  private static byte[] ReadBinaryFile(string aFileName)
  {
    byte[] buf;
    FileStream fs = File.OpenRead(aFileName);
    using (fs)
    {
      int pos = 0;
      int length = (int) fs.Length;
      buf = new byte[length];
      while (true)
      {
        int bytesRead = fs.Read(buf, pos, length-pos);
        if (bytesRead == 0)
        {
          break;
        }
        pos += bytesRead;
      }
    }

    return buf;
  }

  private static void WriteBinaryFile(string aFileName,
    byte[] aFileContents)
  {
    FileStream fs = File.OpenWrite(aFileName);
```

```
    using (fs)
    {
      fs.Write(aFileContents, 0, aFileContents.Length);
    }
  }

  private static string GetImageFormat(string aFileName)
  {
    FileInfo fileInfo = new FileInfo(aFileName);
    string fileExtenstion = fileInfo.Extension;
    string imageFormat = fileExtenstion.ToLower().Substring(1);
    return imageFormat;
  }

  private static int[] ListImageIdsFromDB()
  {
    OleDbConnection dbConn = new OleDbConnection(
      DB_CONNECTION_STRING);
    dbConn.Open();
    using (dbConn)
    {
      OleDbCommand cmd = new OleDbCommand(
        "SELECT ImageId FROM Images", dbConn);
      ArrayList imageIds = new ArrayList();
      OleDbDataReader reader = cmd.ExecuteReader();
      using (reader)
      {
        while (reader.Read())
        {
          int imageId = (int) reader["ImageId"];
          imageIds.Add(imageId);
        }
      }

      int[] imageIdArray = (int[]) imageIds.
        ToArray(typeof(int));
      return imageIdArray;
    }
  }

  private static void ExtractImageFromDB(
    int aImageId, out byte[] aImage, out string aImageFormat)
  {
    OleDbConnection dbConn = new OleDbConnection(
      DB_CONNECTION_STRING);
    dbConn.Open();
    using (dbConn)
    {
      OleDbCommand cmd = new OleDbCommand(
        "SELECT Image, ImageFormat FROM Images " +
```

```
        "WHERE ImageId=@id", dbConn);
      OleDbParameter paramId = new OleDbParameter(
        "@id", OleDbType.Integer);
      paramId.Value = aImageId;
      cmd.Parameters.Add(paramId);
      OleDbDataReader reader = cmd.ExecuteReader();
      using (reader)
      {
        if (reader.Read())
        {
          aImage = (byte[]) reader["Image"];
          aImageFormat = (string) reader["ImageFormat"];
        }
        else
        {
          throw new Exception(
            String.Format("Invalid image ID={0}.", aImageId));
        }
      }
    }
  }

  private static void InsertImageToDB(byte[] aImage,
    string aImageFormat)
  {
    OleDbConnection dbConn = new OleDbConnection(
      DB_CONNECTION_STRING);
    dbConn.Open();
    using (dbConn)
    {
      OleDbCommand cmd = new OleDbCommand(
        "INSERT INTO Images ([Image], ImageFormat) " +
        "VALUES (@image, @imageFormat)", dbConn);

      OleDbParameter paramImage =
        new OleDbParameter("@image", OleDbType.Binary);
      paramImage.Value = aImage;
      cmd.Parameters.Add(paramImage);

      OleDbParameter paramImageFormat =
        new OleDbParameter("@imageFormat", OleDbType.Char);
      paramImageFormat.Value = aImageFormat;
      cmd.Parameters.Add(paramImageFormat);

      cmd.ExecuteNonQuery();
    }
  }

  private static void DeleteAllImagesFromDB()
  {
```

```
        OleDbConnection dbConn = new OleDbConnection(
          DB_CONNECTION_STRING);
        dbConn.Open();
        using (dbConn)
        {
          OleDbCommand cmd = new OleDbCommand(
            "DELETE FROM Images", dbConn);
          cmd.ExecuteScalar();
        }
    }

    static void Main()
    {
        DeleteAllImagesFromDB();
        Console.WriteLine("Deleted all images from the DB.");

        byte[] image = ReadBinaryFile(SOURCE_IMAGE_FILE_NAME);
        string imageFormat = GetImageFormat(
          SOURCE_IMAGE_FILE_NAME);
        Console.WriteLine("Loaded image file {0}.",
          SOURCE_IMAGE_FILE_NAME);

        InsertImageToDB(image, imageFormat);
        Console.WriteLine("Inserted an image in the DB.");

        int[] imageIds = ListImageIdsFromDB();
        Console.WriteLine("There are {0} images in the DB.",
          imageIds.Length);

        int firstImageId = imageIds[0];
        byte[] imageFromDB;
        string imageFormatFromDB;
        ExtractImageFromDB(firstImageId,
          out imageFromDB, out imageFormatFromDB);
        Console.WriteLine("Extracted first image from the DB.");

        WriteBinaryFile(DEST_IMAGE_FILE_NAME, imageFromDB);
        Console.WriteLine("Image saved to file {0}.",
          DEST_IMAGE_FILE_NAME);
    }
}
```

Как работи примерът?

Примерът започва с изтриването на всички изображения от базата. След това прочита съдържанието на един графичен файл в масива от байтове **`image`** и от името му извлича неговото разширение (**`gif`**, **`png`**, **`jpg`**, ...). За целта се използват методите **`ReadBinaryFile(…)`** и **`GetImageFormat(…)`**.

След това с метода **InsertImageToDB(…)** прочетеният файл се вмъква в базата данни в ред от таблицата **Images**. Съдържанието на файла се записва в колоната **Image**, а типът му – в колоната **ImageFormat**.

Следва извличане на списък от идентификаторите на всички графични обекти от таблицата **Images** чрез метода **ListImageIdsFromDB(…)** – в нашия случай обектът е само един.

От списъка се взима първият идентификатор и съответният му графичен обект се извлича от базата в масива от байтове **imageFromDB** чрез метода **ExtractImageFromDB(…)**.

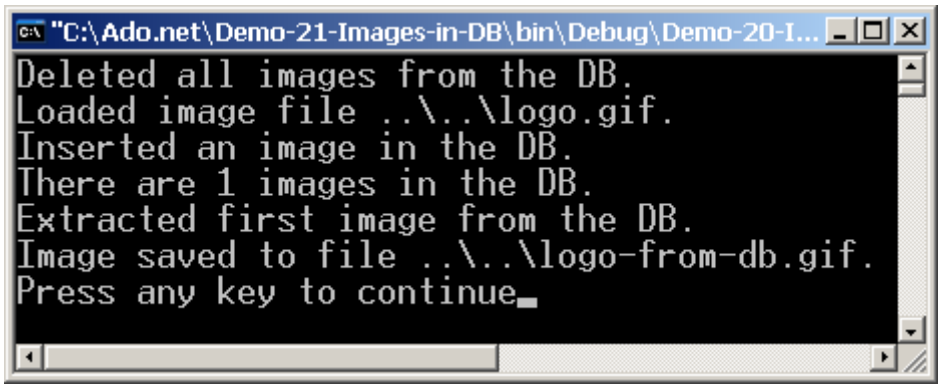
Методът **InsertImageToDB(…)** вмъква графичен обект в базата данни. Той приема като параметри масив от байтове, съдържащ графичния обект и символен низ, съдържащ неговия формат. Методът използва параметрична заявка за добавяне на записа в таблицата **Images**.

Методът **ExtractImageFromDB(…)** извлича от базата графичен обект и формата му. Той приема като входен параметър идентификатора на графичния обект (първичния ключ от таблицата **Images**), а като изходни параметри – масив от байтове, в който ще се съхрани графичният обект след прочитането му и символен низ, който да съхранява съответстващия му формат. Използва се параметрична заявка, резултатът от която се прочита в **OleDbDataReader** обект и се записва в двата изходни параметъра.

Методът **ListImageIdsFromDB(…)** прочита идентификаторите на графичните обекти от базата данни и ги съхранява в списък. След това ги преобразува в стойности от тип **int** и ги връща като масив.

В примера се използват и няколко помощни метода: **ReadBinaryFile(…)**, **WriteBinaryFile(…)** и **GetImageFormat(…)**. Те съответно служат за прочитане в масив от байтове съдържанието на файл, зададен с името си, записване съдържанието на масив от байтове в двоичен файл със зададено име и извличане разширението по дадено име на файл.

Ето и резултатът от изпълнението на примера:



Работа с големи обеми двоични данни

В предходния пример всички файлове, прочетени от диска и извлечени от базата данни, се съхраняват в паметта в масив от байтове. При големи файлове това може да създаде сериозни проблеми, например ако някой се опита да запише в базата данни филм с обем 700 MB. В такъв случай приложението или ще изразходва прекалено много памет и ще работи изключително бавно, или просто ще настъпи срив.

За да се избегнат подобни проблеми, е важно, когато се работи с големи по обем двоични данни, четенето и записът им да стават последователно на части, а не наведнъж.

Четене на обемни данни

Четенето на обемни данни от **SqlDataReader** може да се извърши като се замени извличането на стойността чрез свойството **Value** с поредица от извличания на части от нея с метода **GetBytes(…)**. Преди това, обаче, е необходимо при изпълнението на **ExecuteReader()** да му се подаде параметър **CommandBehaviour.SequentialAccess**, с което да се укаже, че данните ще бъдат извличани на части. При използване на последователен достъп в **SqlDataReader** е неодходимо колоните да се четат една след друга в реда, в който са върнати от SQL заявката.

Запис на обемни данни

Записът на обемни двоични данни на части в базата данни е малко полесен. Класът **SqlParameter** има свойство **Offset**, чрез което може да се задава отместването в даден запис при вмъкване на двоични данни. Чрез него може вмъкването на обемен двоичен файл да се замени с поредица вмъквания от по 64 KB с подходящо отместване.

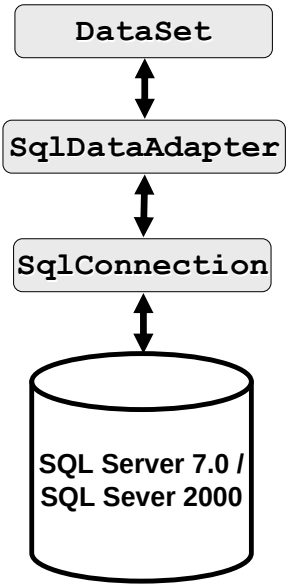
ADO.NET в несвързана среда

Както вече беше обяснено, най-характерното за несвързания модел е, че се работи с копие на данните от базата, което се намира в паметта на локалната машина. Докато се данните се обработват локално, не се поддържа отворена връзка към сървъра на базата от данни.

Първоначално се отваря връзка към базата данни (**SqlConnection**) и част от данните се зареждат и кешират в паметта в **DataSet** обект и връзката се преустановява. **DataSet** обектите съдържат съвкупност от таблици и връзки между тях. Ще се спрем на тях в детайли след малко.

След обработка на заредените данни в паметта, е възможно повторно отваряне на връзка за внасяне на промените, които потребителят е направил. Това става най-често с класа **SqlDataAdapter**, който също ще разгледаме в детайли по-късно.

На картинката е показано взаимодействието на споменатите три класа със сървъра и помежду им:



Работата с данни в несвързана среда може да се опише по-подробно със следната базова последователност от стъпки (даденият пример е за SQL Server Data Provider, но същата последователност е в сила и за останалите доставчици на данни):

1. Отваряне на връзка (**`SqlConnection`**).
2. Създаване и запълване на **`DataSet`** обект (чрез **`SqlDataAdapter`** обект).
3. Затваряне на връзката.
4. Работа с **`DataSet`** обекта (на тази стъпка се извършва необходимата обработка върху заредените данни).
5. Отваряне на връзка.
6. Нанасяне на промените от **`DataSet`** обекта върху данните в базата от данни (и разрешаване на конфликтите, когато има такива).
7. Затваряне на връзката.

Преди да преминем към разглеждане на класовете **`DataSet`** и **`DataAdapter`**, нека първо си изясним кога се използва несвързаният модел за работа с данни и кога е по-подходящо да се използва свързаният.

Типични сценарии за работа в несвързана среда

Несвързаният модел се използва, когато не е необходима постоянна връзка с базата от данни и потребителите трябва само от време на време да извличат данни и да правят промени в базата данни.

Типичният случай за използване на несвързан модел е при уеб приложенията. При тях не е нормално да се държи отворена връзка между две последователни клиентски заявки, защото те може да са много раздалечени във времето и няма гаранция в кой момент клиентът ще поиска някакви данни или ще поиска да направи някаква промяна в базата данни.

От гледна точка на ефективност, ако приложението се използва от много потребители, които едновременно достъпват базата, и извършва продължителна обработка върху данните, е за предпочитане използването на несвързан модел. По този начин се спестяват ресурси на сървъра на базата и става възможно обслужването на много потребители. Транзакциите траят много кратко време и така не се налага потребителите да се изчакват дълго един друг.

Друг сценарий на използване на несвързания модел е интеграцията на данни от различни източници. В **`DataSet`** данните се представят по един и същ начин, независимо дали идват от база от данни или от XML файл или от друг източник. Няма значение и от каква точно база от данни са извлечени. Така е възможно едно приложение да използва данни, които се съхраняват в бази на различни производители.

Важно приложение на несвързания модел е в изграждането на многослойни приложения. Ако приложението използва бизнес обекти за достъп до данните и те са разположени на сървър, обслужващ междинен слой, то бизнес обектите трябва да предават несвързани структури от данни на клиентския слой на приложението. Това лесно се осъществява чрез **`DataSet`** обект – съдържанието му лесно се предава между различни компоненти.

Несвързан модел в ADO.NET, XML и уеб услуги

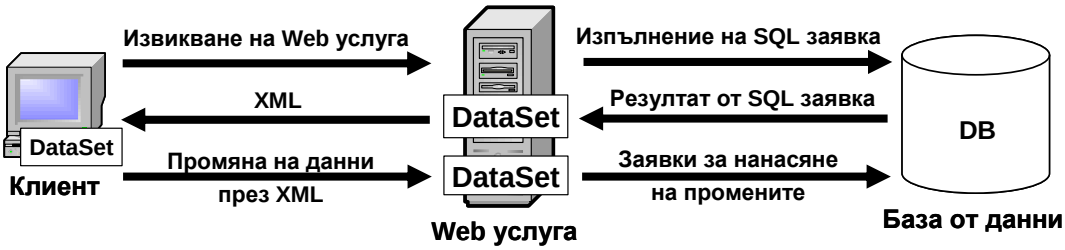
Една от най-важните характеристики на ADO.NET е тясната му интеграция с XML. В него са вградени много средства, които улесняват работата с данни в XML формат и трансформацията на данни, извлечени от релационна база данни към XML формат.

Тези възможности на ADO.NET, заедно с богатите средства за работа с несвързани данни, които предлага, се използват често при реализацията на уеб услуги. Уеб услугите са базирани на отворени стандарти (XML, SOAP, WSDL и др.), благодарение на което те могат да се използват за междуплатформена интеграция или в хетерогенна среда каквато е Интернет. Уеб услугите ще разгледаме в детайли в темата "ASP.NET уеб услуги".

Ето един типичен пример за интеграцията на ADO.NET с уеб услуги чрез несвързания модел за достъп до данни:

Имаме трислойна архитектура. Бизнес слоят е реализиран чрез ASP.NET уеб услуга, back-end слоят е реализиран с база данни, а front-end слоят е настолно или уеб приложение, което използва уеб услугата.

Благодарение на интеграцията между ADO.NET с XML стандарта, данни, извлечени от база от данни в **DataSet** обекти, могат да бъдат предавани като XML документи между слоевете на приложението. Следната диаграма илюстрира тази схема на работа:



Когато клиентът поиска данни, се обръща към уеб услугата, а тя ги извлича от базата данни във вид на **DataSet** обект. След това услугата ги транспортира във вид на XML и клиентът получава извлеченият **DataSet** обект. След това клиентът работи с извлечените данни известно време и променя част от тях. Накрая изпраща промените към уеб услугата отново като **DataSet** обект (който се транспортира във вид на XML). Услугата нанася промените в базата данни и евентуално разрешава конфликтите, ако има такива.

Класове за достъп до данните в несвързана среда

За реализация на несвързания достъп до данни се използват набор от класове, които са дефинирани в пространството от имена **System.Data** (освен **SqlDataAdapter**, който е дефиниран в **System.Data.SqlClient**). Повечето от тях (всъщност, всички без класа **SqlDataAdapter**) са независими от доставчика на данни. Следва кратко описание на основните класове от несвързания модел на ADO.NET:

- **DataSet** – основен клас за представяне на данните в паметта. Той може да се разглежда като абстракция на релационна база от данни, състояща се от таблици с връзки между тях. По същество **DataSet** е контейнерен клас, който съдържа таблици (**DataTableCollection**), релации (**DataRelationCollection**), ограничения (които са част от **DataTableCollection**) и някои други класове.

- **DataTable** – клас, представящ таблица в паметта. Един **DataSet** може да съдържа множество такива обекти, като чрез тях представя цяла база от данни в паметта. **DataTable** съдържа обекти за колони (**DataColumn**), редове (**DataRow**) и ограничения (**Constraints**).

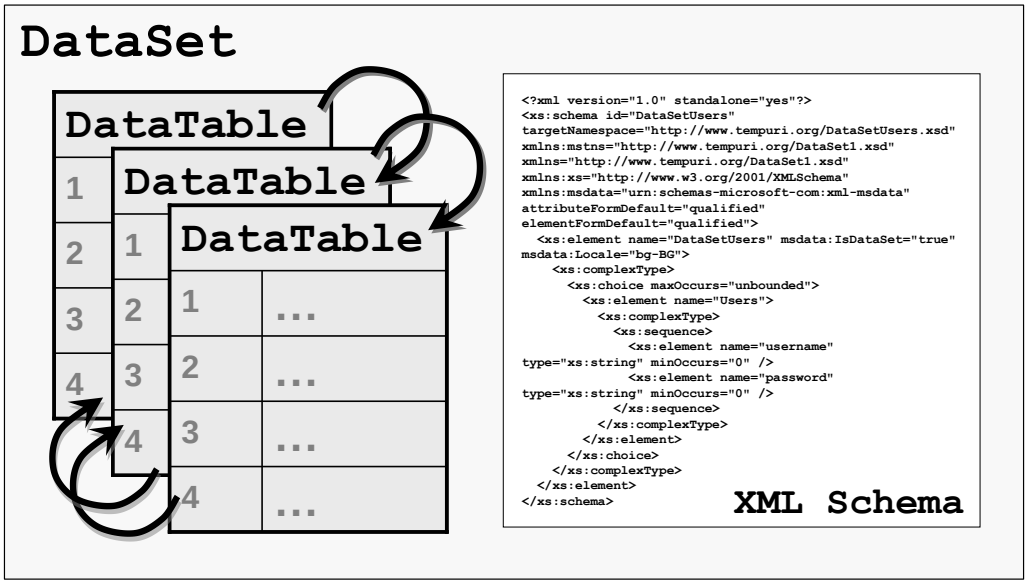
- **XxxDataAdapter** –осъществява достъпа до данните като използва **XxxCommand** и **XxxConnection** класове. Чрез този клас става запълването на **DataSet** обектите с данни. Той функционира като "мост" между базата и **DataSet** обектите. Чрез него се извършва и обновяването на данните в базата след промяна в паметта. Този клас

зависи от доставчика на данни и има различни версии за различните доставчици, например **SqlDataAdapter**, **OleDbDataAdapter** и др.

- **DataRelation** – представя връзка между таблици, съхранявани в **DataSet** обект.

DataSet – обектен модел

Вече се запознахме с основните класове от несвързания модел на ADO.NET. Сега ще разгледаме най-важният клас от него – класът **DataSet**. Ето опростена схема на неговия обектен модел:



Колекции в DataSet

Класът **DataSet** съдържа съвкупност от таблици и връзки между тях. Те са представени чрез две основни колекции – **Tables** и **Relations**.

Свойството **DataSet.Tables** е обект от тип **DataTableCollection** и съдържа един или повече обекта от тип **DataTable**. Всеки обект **DataTable** представя таблица от източника на данни.

DataSet.Relations е свойство, което е от тип **DataRelationCollection** и съдържа един или повече обекти от тип **DataRelation**. Те представят връзка от тип родител-наследник (master-detail) между две таблици от базата, като обикновено се базират на стойността на външен ключ.

Схема на DataSet

Схемата на един **DataSet** обект описва цялата му структура (подобно на релационните схеми в базите данни). Тя включва всички таблици (заедно с техните колони, ограничения и т. н.) и връзките между таблиците в

него. Схемата може да се представи по два начина, което разграничава и два различни вида **DataSet** обекти – силно типизиран и нетипизиран.

При силно типизирания **DataSet** схемата е описана във външен XML Schema (**.xsd**) файл.

Възможно е схемата да се зададе и програмно, като отделните таблици и колони в тях се присвоят на **DataSet** обекта по програмен път. Такъв **DataSet** е нетипизиран. Той няма съответна вградена схема (файл, описващ схемата), но структурата му може да бъде извлечена от него.

И двата вида **DataSet** обекти могат да се използват, но Visual Studio предлага повече средства за работа със силно типизиран **DataSet**, които улесняват програмирането и намаляват вероятността от грешки.

Схемата на **DataSet** обектите, както и таблиците с данните в тях, могат да се извличат и да се записват във вид на XML документи. Тази възможност широко се използва при запазване на **DataSet** обекти във вид на XML, транспортиране и последващо възстановяване.

Силно типизирани DataSets

Силно типизираните **DataSets** наследяват класа **DataSet** и използват информацията от XML Schema файл, за да генерират нов клас. В този наследен клас обектите в колекциите, представящи таблиците, колоните, редовете и т.н. наследяват съответно от **DataTable**, **DataColumn**, **DataRow** и т.н., като се добавят специфични методи, свойства и събития съобразно използваната схема.

Например, ако използваме XML Schema файл за да генерираме типизиран **DataSet** и в него се описва таблица **MyTable**, то генерираният типизиран **DataSet** клас ще има свойство **MyTable** за директен достъп до таблицата със съответното име. При нетипизирания **DataSet** няма възможност за директен достъп по този начин.

Използването на типизирани **DataSets** прави програмирането по-интуитивно. То позволява на Visual Studio да предложи например функционалност като autocomplete, а на компилатора да проверява за несъответствие на типовете и други грешки още по време на компилация, вместо по време на изпълнение.

Следват два примера, които илюстрират разликата между достъпа до нетипизиран и силно типизиран **DataSet**.

Достъп до нетипизиран DataSet

Ето типичен пример за достъп до нетипизиран **DataSet** обект:

```
DataSet dsUsers = ...;
string username = (string) dsUsers.
  Tables["Users"].Rows[0]["username"];
```

Виждаме, че за да осъществим достъп до стойността на полето **username** в първия ред на таблицата **Users** трябва да използваме колекциите **Tables** и **Rows** и да индексираме съответно по имената на таблицата и полето (което е има на колона в таблицата), които са ни нужни. Освен това трябва да преобразуваме типа на върнатата стойност към **string**.

Достъп до силно типизиран DataSet

Ето типичен пример за достъп до силно типизиран **DataSet**:

```
UsersDataSet dsUsers = ...;
string username = dsUsers.Users[0].username;
```

Този пример прави същото, което прави и горният, но в него е използван силно типизиран **DataSet**. Както виждаме, не използваме явно колекциите **Tables** и **Rows**. Вместо това имаме свойство **Users**, което дава достъп до съответната таблица, имаме свойство **username**, което представя колона на таблицата, а също така и типът на стойността е **string** и не е нужно да се извършва конвертиране. В този случай достъпът до данните в **DataSet** обекта е значително улеснен.

Създаване на DataSet

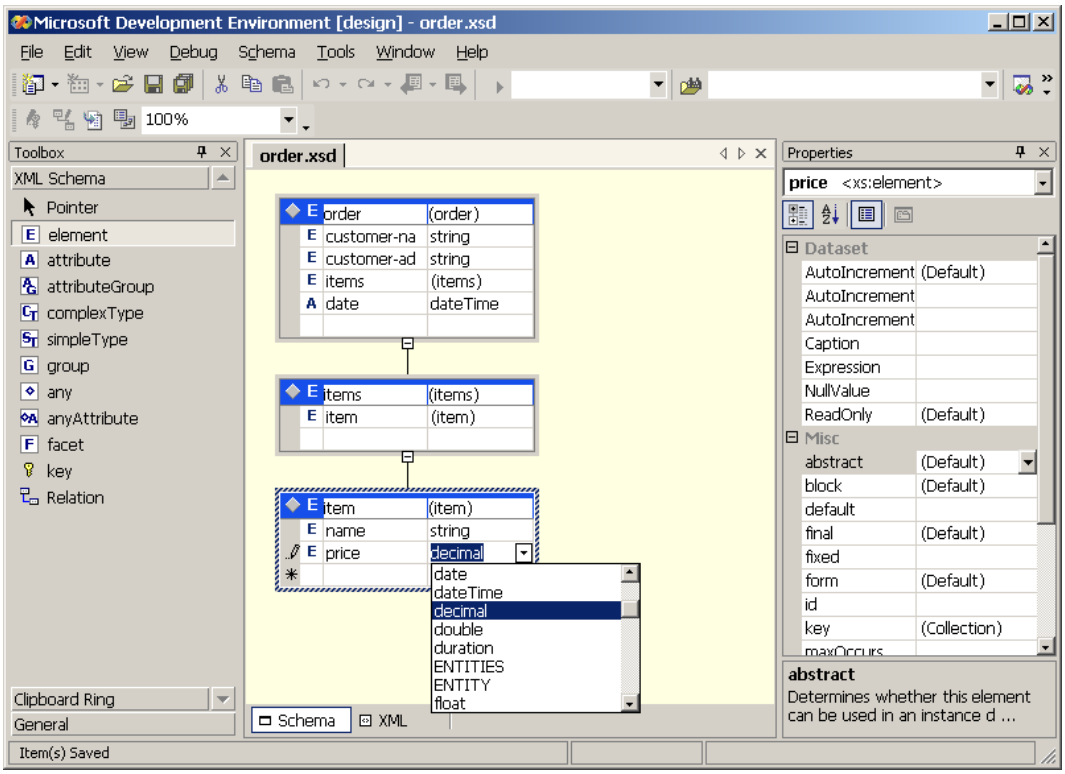
Ще разгледаме два начина за създаване на силно типизиран **DataSet** – чрез XSD дизайнера на Visual Studio .NET и с инструмента **xsd.exe**.

Създаване на силно типизиран DataSet през XML дизайнера

Можем да използваме XML дизайнера на VS.NET, ако искаме да имаме фин контрол над дефинирането на схемата или ако дефинираме схема, която не се базира на външен източник.

Дизайнерът се отваря, когато добавим **DataSet** обект чрез подменюто **Add New Item** на менюто **File**. По-точно, при избор на **File | Add New Item** се отваря диалогов прозорец, в лявата част на който има изброени различни шаблони на обекти, които могат да бъдат добавяни в приложението. Оттам трябва да изберем шаблона **Data Set**. В резултата на това се отваря **.xsd** файл, в който можем да опишем схемата на **DataSet**. Можем сами да опишем цялата схема, а можем и да влачим и пускаме таблици от прозореца на **Server Explorer** и след това да редактираме генерираната автоматично схема според нуждите си.

При запазване на **.xsd** файла автоматично се генерира съответният C# клас. Ако сме правили промени в **.xsd** файл, съответстващ на съществуващ силно типизиран **DataSet** при запазване на промените във файла се генерира отново и съответния клас, за да бъдат отразени направените промени.

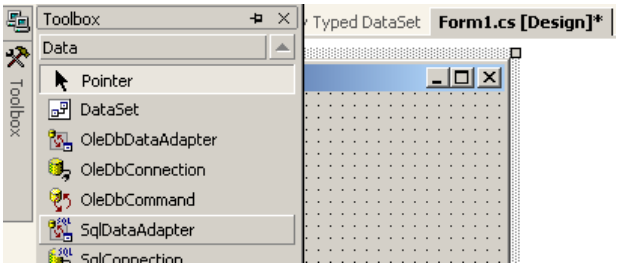


Създаване на силно типизиран DataSet с VS.NET – пример

Ще демонстрираме още един начин за създаване на силно типизиран **`DataSet`** – чрез VS.NET.

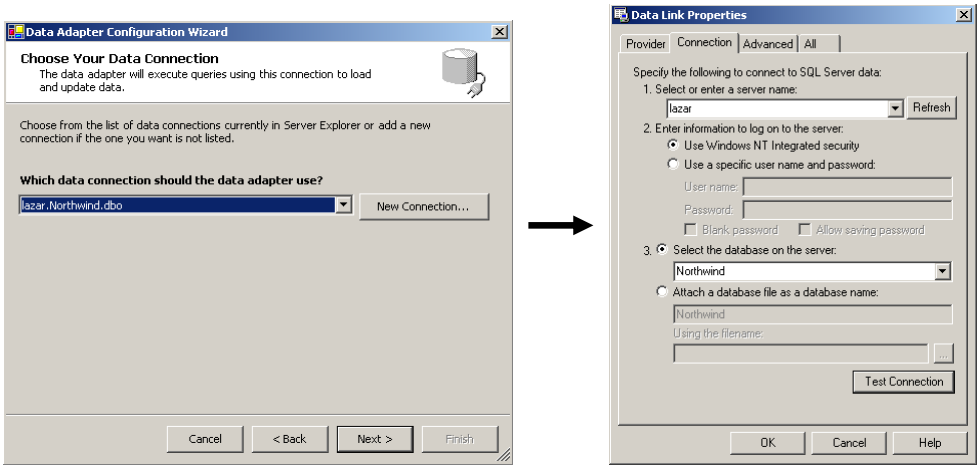
1. Създаваме Windows Application проект с име **`StronglyTypedDataSet`** (от менюто **`File | New | Project`**).

2. След създаване на проекта се отваря дизайнерът на VS.NET, в който има празна форма. С влачене и пускане поставяме **`SqlDataAdapter`** обект от Data страницата на Toolbox върху формата:



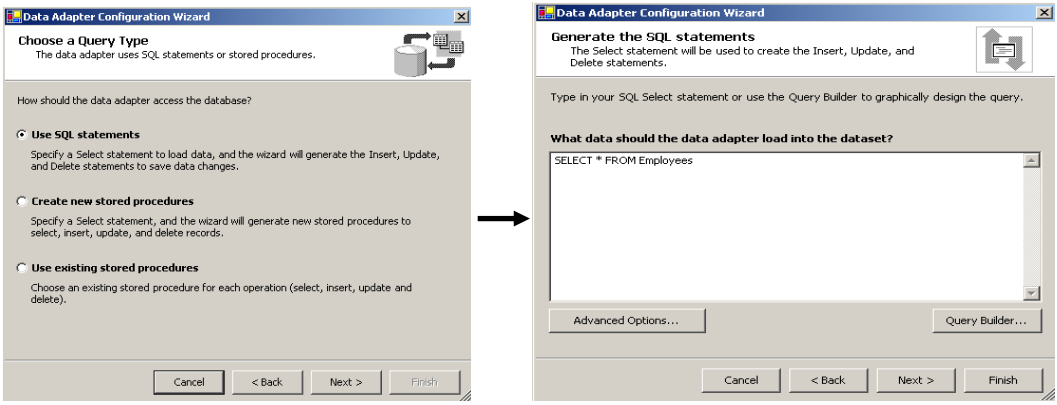
3. При пускането на **`SqlDataAdapter`** обекта се стартира помощникът за конфигуриране на **`DataAdapter`** (DataAdapter configuration wizard). Избираме бутона [Next] на прозореца на помощника. Отваря се друг прозорец, където можем да изберем вече съществуваща връзка към базата, която адаптерът да ползва, или да създадем нова.

4. Избираме бутона New Connection… От следващия прозорец задаваме настройки за връзката – избираме сървъра на базата от данни, методът на автентикация и името на базата, с която ще работим. Избираме Windows автентикация и база **`Northwind`**:

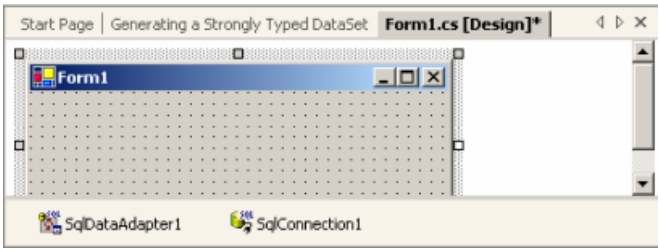


5. От следващия прозорец се избира вида на заявката, която адаптерът ще използва, за да запълни бъдещия **`DataSet`**. Избираме SQL Statements и натискаме [Next].

6. В следващия прозорец задаваме заявката, с която ще работи адаптерът. В случая ще извлечем съдържанието на таблицата **`Employees`** чрез SQL заявката "**`SELECT * FROM Employees`**":



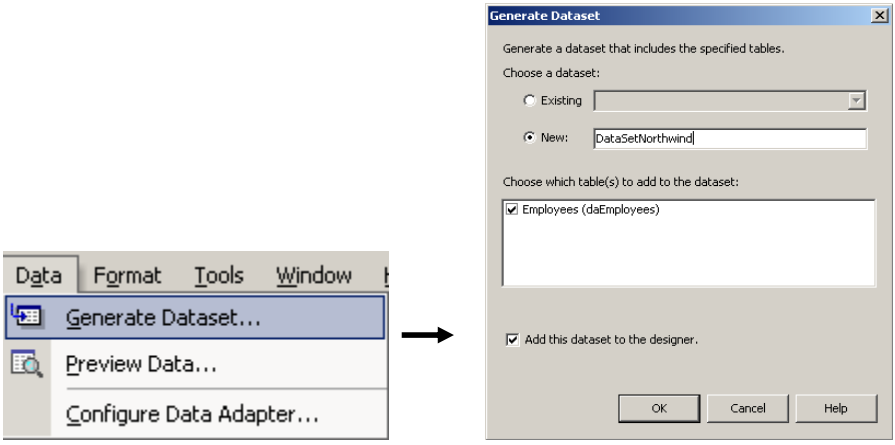
7. След въвеждане на заявката натискаме [Next] и в прозореца, който се отвори избираме [Finish]. Обектите на връзката и адаптера се появяват в поле в долната част на VS.NET дизайнера (нарича се Component Designer). Те имат подразбиращи се имена – съответно **`SqlConnection1`** и **`SqlDataAdapter1`**:



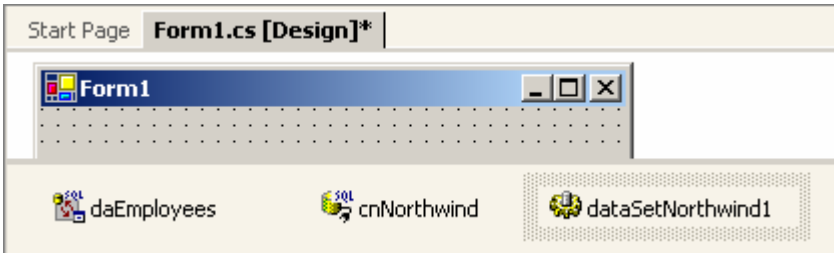
8. Избираме обекта на връзката в прозореца Properties в ляво и променяме името му на **cnNorthwind**. По същия начин променяме името на адаптера на **daEmployees**.

9. Избираме **daEmployees** и стартираме генериране на **DataSet** от Data | Generate DataSet.

10. В прозореца, който се отваря, оставяме настройките по подразбиране. Те указват да се генерира нов типизиран **DataSet**, който да съдържа таблицата **Employees** и да бъде добавен към дизайнера. Променяме името на **DataSet** класа, който ще се генерира, от **DataSet1** на **DataSetNorthwind**.

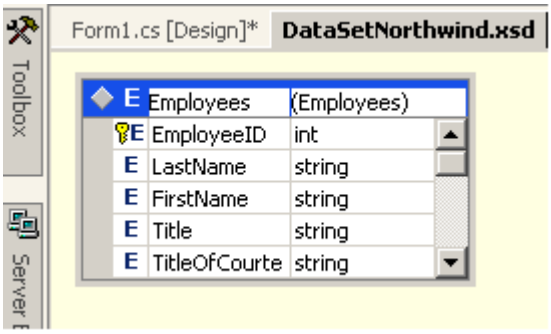


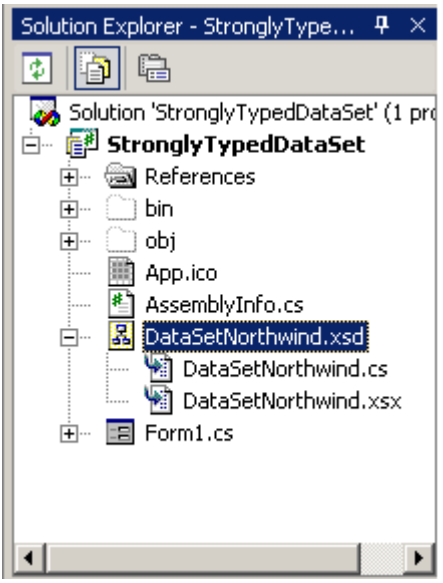
11. След като изберем [OK], в горния прозорец се генерира строго типизирания **DataSet DataSetNorthwind** (той е клас) и се добавя инстанция **dataSetNorthwind1** на този клас към дизайнера:



12. Ако сега отворим от Solution Explorer файла, **DataSetNorthwind.xsd**, ще видим, че той съответства на схемата на таблицата **Employees** от

базата данни, а зад него стои класът **DataSetNorthwind**, който се намира в автоматично генерирания от Visual Studio .NET файл **DataSetNorthwind.cs**.





Можем да разгледаме сорс кода на класа **DataSetNorthwind** и ще установим, че той съдържа свойство за достъп до таблицата **Employees**, която е от тип **EmployeesDataTable**. Този клас също е силно типизиран и съдържа свойства за достъп до полетата и редовете от **Employee** таблицата.

Създаване на силно типизиран DataSet чрез xsd.exe

Друг начин за създаване на силно типизиран **DataSet** е чрез използване на инструмента **xsd.exe**, който се предоставя заедно с .NET Framework SDK. За да използваме инструмента трябва да разполагаме с XSD файл, описващ схемата на **DataSet**, за който искаме да създадем строго типизирани класове за достъп.

За генериране на **DataSet** чрез командата **xsd.exe** се използва следният синтаксис:

```
xsd.exe /dataset MyDataSet.xsd
```

Резултатът от изпълнение на командата е файл **MyDataSet.cs**, който може да бъде компилиран и да се използва в ADO.NET приложение. Генерираният **DataSet** клас съдържа специални свойства за таблиците от схемата и за техните редове и колони.

Поддръжка на автоматично свързване

DataSet е един от класовете в .NET Framework, които поддържат автоматично свързване (data binding) към Windows Forms контроли и ASP.NET уеб контроли.

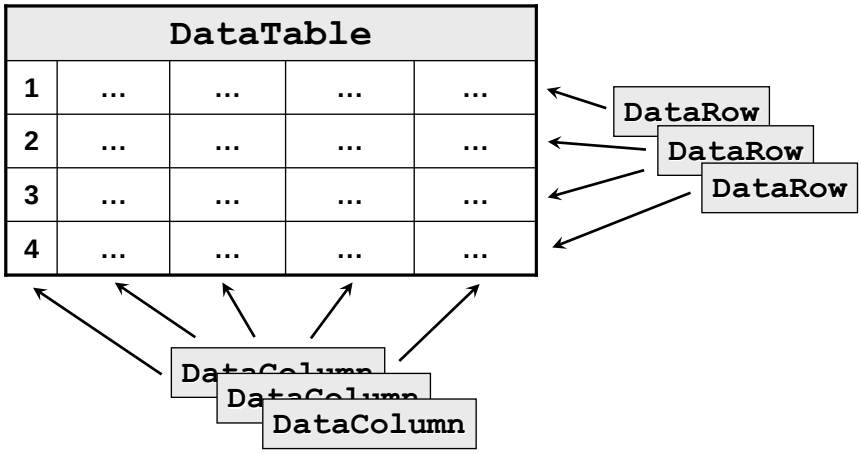
Data Binding технологията осигурява автоматично задаване на стойности на свойства на една или повече контроли по време на изпълнение. Стойностите се взимат от някаква структура от данни (масив, списък, таблица и т.н.).

При **DataSet** е възможно различни контроли да се свържат към различни полета на една таблица, или към различни таблици в **DataSet** обекта. Например, възможно е две **DataGrid** контроли да се свържат към две различни таблици в **DataSet**, като представят информацията от двете таблици съответно. Възможно да се осъществяват връзки между отделните контроли. Така могат да се реализират master-detail зависимости между различни таблици. За целта е нужно да бъдат създадени и обекти за съответните релации в **Relations** колекцията на **DataSet** обекта. Повече за автоматичното свързване може да прочетете в темата "Графичен потребителски интерфейс с Windows Forms".

Класът DataTable

Класът **DataTable** съхранява данните, подобно на таблица в базата от данни. Той представлява кеширано копие на данните от таблица в паметта. Този клас има важна роля в архитектурата на ADO.NET. **DataTable** може да се използва както самостоятелно, така и като част от **DataSet** обекти. Използва се за тип на елементите от колекцията **Tables** на **DataSet** класа.

Една **DataTable** таблица се описва от колони (**DataColumn**), а данните в нея се съхраняват във вид на редове (**DataRow**):



DataTable дава възможност за извършване на различни операции върху данните, които съхранява. Данните могат да бъдат филтрирани, сортирани и модифицирани.

DataTable съдържа три колекции – **Columns**, **Rows** и **Constraints**. Колекциите **Columns** и **Constraints** дефинират схемата на таблицата, докато **Rows** съдържа самите данни.

DataTable поддържа списък на всички промени

DataTable таблиците са по-сложни, отколкото изглежда на пръв поглед. Те не само съдържат в себе си съвкупност от редове със стойности, но и пазят всички промени, направени по тях. Това много улеснява идентифицирането на промените, които са извършени в дадена таблица от момента на нейното зареждане.

Всяка таблица поддържа списък от добавените, изтритите и променените редове в нея. Този списък при първоначално е празен. Той може да бъде извлечен по всяко време с метода **GetChanges()**, а при нужда може да бъде изтрит с метода **AcceptChanges()**.

Типичен сценарии за работа при несвързания модел за достъп до данните е следният:

1. Свързваме се с базата данни и зареждаме данни в дадена таблица (или група от таблици, съдържащи се в даден **DataSet**).
2. Работим локално с таблицата – добавяме, редактираме и изтриваме редове от нея.
3. В даден момент се свързваме отново с базата данни и я актуализираме. Това става като извлечем променените записи с метода **GetChanges()** и извършим съответни промени, като разрешим евентуалните конфликти, които се получават. Накрая извикваме метода **AcceptChanges()** за да анулираме списъка с промените по таблицата, тъй като те вече са отразени в базата данни.

Нанасянето на промените в базата данни и разрешаването на конфликтите, които могат да се получат, е сложна задача, която се извършва най-често посредством **DataAdapter** класовете. Този проблем ще разгледаме в детайли малко по-нататък.

Основни свойства на DataTable

Както знаем, данните в класа **DataTable** са съставени от редове, всеки от които представлява съвкупност от полета, а типовете на тези полета се дефинират от колоните. За достъп до редовете, колоните и останалите елементи на една таблица класът **DataTable** предлага съответни свойства. Да разгледаме по-важните от тях:

- **Rows** – връща колекция от редовете в таблицата. Обектите в колекцията са от тип **DataRow** и могат да се достъпват чрез индексиране по номер на реда – **Rows[index]**.
- **Columns** – представя колекция от колоните в таблицата. Обектите в тази колекция са от тип **DataColumn** и могат да бъдат достъпвани чрез индексиране по индекс или по име – **Columns[…]**.
- **PrimaryKey** – използва се за задаване и извличане на първичен ключ на таблицата. Съдържа масив от **DataColumn** обекти, които съставят първичния ключ на таблицата. Задава уникална стойност и по него могат да се търсят редове в таблицата.
- **Constraints** – използва се за задаване на ограничения в таблицата. Ограниченията могат да бъдат по уникалност на колона (или група от колони) или по външен ключ.
- **HasErrors** – връща дали в таблицата има грешки. Грешките се използват най-често за индикация на проблеми, свързани с нанасянето на промените в базата данни.

Основни методи на DataTable

По-важните методи на класа DataTable са свързани с добавянето на нови редове и колони:

- **NewRow()** – създава нов празен ред (**DataRow**), който съдържа полета, съответстващи на колоните от таблицата, но не го добавя в таблицата. Програмистът трябва явно да го добави, евентуално след като е задал стойности на полетата му.
- **Rows.Add(DataRow)** – добавя ред към колекцията **Rows** на таблицата. Редът трябва да е бил вече създаден преди това.
- **Columns.Add(DataColumn)** – добавя колона към колекцията от колони на таблицата. Колоната трябва да е била създадена преди това.
- **GetChanges()** – извлича списъка на промените, направени по таблицата.
- **AcceptChanges()** – нанася всички промени по данните в таблицата и изчиства списъка на промените.
- **GetErrors()** – връща редовете с грешки. Използва се най-често за извличане на списъка със записите, които са предизвикали конфликт при обновяване на базата данни.

Работа с DataTable

В тази точка ще навлезем в повече детайли относно работата с класа **DataTable**, ще разгледаме класовете **DataRow** и **DataColumn** и събитията, които **DataTable** поддържа.

Създаване на DataTable

В създаването на инстанции на **DataTable** няма нищо сложно. Създават се както всички обекти в .NET Framework:

```
DataTable table = new DataTable();
```

След създаването на дадена таблица, обикновено се дефинират колоните ѝ чрез добавяне на **DataColumn** инстанции към списъка ѝ **Columns**. След това обикновено в таблицата се зареждат някакви редове. Ще видим пример как става всичко това след малко.

Класът DataRow

Колекцията **Rows** на класа **DataTable** съдържа обекти от клас **DataRow**. Класът **DataRow** представлява един ред от таблица. Чрез него може да се извличат и променят стойностите в полетата на реда. **DataRow** има по едно поле за всяка колона от колекцията **Columns** на таблицата, с която е свързан.

Понеже, както вече знаем, таблиците запомнят всички промени, направени по техните редове. По тази причина редовете съхраняват информация за състоянието си. Един ред може да бъде в различни състояния – променен, изтрит, новодобавен, разкачен (несвързан с таблица) или непроменен. Състоянието на даден ред се съхранява в свойството му **RowState**, което е от изброения тип **DataRowState**.

Създаване на редове

Конструкторът на класа **DataRow** нарочно е направен частен, за да не се извиква директно. Обект от клас **DataRow** се създава с извикване на метода **NewRow()** на класа **DataTable**. Този метод създава празен ред според схемата на таблицата без да го добавя в нея. Новосъздадените редове първоначално са в състояние "разкачен", защото не са добавени към никоя таблица.

След като е създаден редът, можем да задаваме стойности на полетата му и да го добавим към колекцията **Rows** на съответния **DataTable** обект.

Достъпът до стойност в дадена колона на реда се осъществява чрез индексиране, което може да бъде по номер или по име на колона.

Класът **DataRow** има метод **Delete()**, който маркира реда като изтрит. Този метод не изтрива реда от **DataTable**, докато не се извика методът **AcceptChanges()** за този ред, или **Update()** метода на **DataAdapter** обекта (всъщност, последният метод имплицитно вика **AcceptChanges()** за реда).

Добавяне на ред в таблица – пример

Ето един пример за създаване и добавяне на ред в таблица, която съдържа информация за автори:

```
// The row is detached (not added to the table)
DataRow row = authorsTable.NewRow();

row["au_id"] = 1;
row["au_fname"] = "Branimir";
row["au_lname"] = "Giurov";
row["au_phone"] = "+359 2 XXX XXXX";

// Add the row to the table
authorsTable.Rows.Add(row);
```

Класът **DataTable** съдържа колекция от редове и не позволява една и съща инстанция на референтния тип **DataRow** да бъде добавяна повече от веднъж.

За всеки нов ред, който ще добавяте към таблица трябва да създадете нов обект от тип DataRow, а не да използвате повторно някой вече създаден. В противен случай ще ще настъпи изключение.

Класът DataColumn

Колекцията **Columns** на класа **DataTable** съдържа **DataColumn** обекти. **DataColumn** описва една колона от **DataTable**. Тя задава името (чрез свойството **ColumnName**), типа на данните в колоната (свойство **DataType**), както и информация за това дали стойностите в колоната се генерират автоматично, различни ограничения върху стойностите в колоната и др.

Нова колона се създава с оператора **new** като се извика конструктора на класа **DataColumn**. След като се зададат стойности на свойствата на създадения обект, той се добавя към колекцията **Columns** на съответния **DataTable** обект.

Тъй като **DataColumn** е част от несвързания модел, този клас е независим от източника на данни. По тази причина, типът на данните в колоната, който се задава, е .NET Framework тип, а не тип, свързан с източника. При извличане на данните се прави съответствие между зададения тип и върнатия от доставчика на данни тип. Когато се задава типът на колона, трябва да се внимава задаваният тип да е съвместим с типа в източника на данни.

Колони с автоматично генерирани стойности

Възможно е за дадена колона да се зададе стойностите в нея да се генерират автоматично. Така се осигурява уникална стойност в съответната колона за всеки добавен в таблицата ред. За тази цел се използват следните свойства на **DataColumn**:

- **AutoIncrement** – приема стойности **true** и **false**, като указва дали стойностите в колоната да се генерират автоматично.
- **AutoIncrementSeed** – задава началната стойност, от която да започне генерирането на стойности.
- **AutoIncrementStep** – задава число, с което да се различават текущо генерираната стойност от предходната (може да бъде както положително, така и отрицателно).

Ето един пример за създаване на колона, която се увеличава автоматично при добавяне на ред:

```
DataColumn column = new DataColumn("id", typeof(long));
column.AutoIncrement = true;
column.AutoIncrementSeed = 0;
column.AutoIncrementStep = 1;
```

След като колоната е създадена, тя може да бъде добавена в дадена таблица по следния начин:

```
DataTable table = new DataTable();
table.Columns.Add(column);
```

Добавянето на даден обект от тип **DataColumn** към дадена таблица може да става най-много веднъж.

Аналогично на редовете, при създаването на нова колона, която ще добавяте към таблица, трябва да създадете нов обект от тип DataColumn.

Събития на класа DataTable

Класът **DataTable** има четири събития, които се предизвикват от действия, изпълнявани върху редове от таблицата и две събития, свързани с действия върху колоните.

Събитията **RowChanging** и **RowChanged** се предизвикват при промяна на ред или при промяна на свойството **RowState** на даден ред. Това включва добавяне на ред, промяна на поле от реда, изтриване на ред и др. Първото от тях се предизвиква преди промяна, а второто – след. Причината за настъпване на събитието може да се определи от свойството **Action** на аргумента **DataRowChangeEventArgs**, който се подава на събитието. Самият ред, който бива променян, може да се достъпи чрез свойството **Row** на същия аргумент.

Събитията **RowDeleting** и **RowDeleted** настъпват съответно преди и след изтриване на ред от таблицата. Те имат същите аргументи и свойства като предходните две събития.

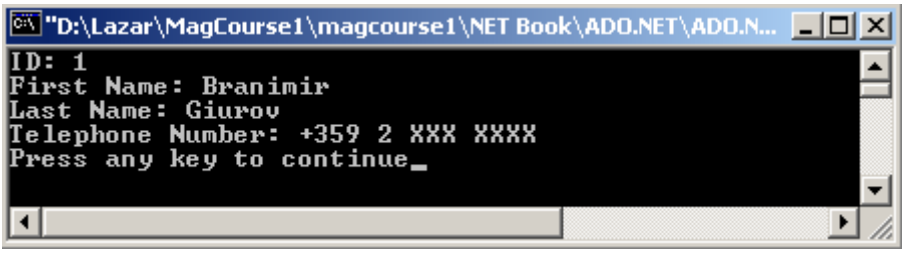
Събитията **ColumnChanging** и **ColumnChanged** се предизвикват при промяна на съдържанието на някое поле в някой ред на таблицата. Първото се предизвиква при промяна на стойност, а второто – след извършване на промяната. И двете събития предоставят на обработчика на събитието аргумент от тип **DataColumnChangeEventArgs**. Този аргумент съдържа специфична информация за събитието, например има свойства **Row** и **Column**, чрез които може да се определи кой ред се променя и в коя колона се извършва промяната. Тези събития често се използват за валидиране на данни.

Работа с DataTable – пример

Следващият фрагмент от сорс код илюстрира използването на **DataTable**:

```
DataTable tbl = new DataTable("Authors");

tbl.Columns.Add("au_id", typeof(int));
tbl.Columns.Add("au_fname", typeof(string));
tbl.Columns.Add("au_lname", typeof(string));
tbl.Columns.Add("au_phone",typeof(string));

// The row is detached (not added to the table)
DataRow row = tbl.NewRow();

row[0] = 1;
row[1] = "Branimir";
row[2] = "Giurov";
row["au_phone"] = "+359 2 XXX XXXX";

tbl.Rows.Add(row);

Console.WriteLine("ID: " + tbl.Rows[0][0] + "\n" +
   "First Name: " + tbl.Rows[0]["au_fname"] + "\n" +
   "Last Name: " + tbl.Rows[0][2] + "\n" +
   "Telephone Number: " + tbl.Rows[0][3]);
```

Резултатът от изпълнението на горния пример е показан на картинката:



В примера се създава нов **DataTable** обект, създават се и се добавят няколко колони, след което се създава нов ред от таблицата, като се използва методът **NewRow()** на **DataTable**. Задават се стойности на полетата на реда и той се добавя към колекцията **Rows** на таблицата. Забеле-

жете, че полетата на реда са достъпни както по индекс, така и по име на колоната. Накрая въведените в реда стойности се извеждат.

Използване на ограничения (constraints)

Заедно с колоните, ограниченията участват в описанието на схемата на таблицата. Нека разгледаме различните видове ограничения, които се поддържат от класа **DataTable**.

Първичен ключ

Първичният ключ представлява една или няколко колони, които уникално определят всеки ред в таблицата. За разлика от останалите видове ограничения, той е достъпен чрез свойството **PrimaryKey** на **DataTable**, а не е част от колекцията **Constraints**. Първичният ключ представлява същевременно и **Unique** ограничение. По него могат да се търсят редове с метода **Find()** на колекцията **Rows**.

Дефиниране на първичен ключ (primary key)

Следващият код показва дефиниране на първичен ключ. Демонстрира се дефиниране на първичен ключ, съставен от една колона и от няколко колони:

```
// Single column PK
dtCustomers.PrimaryKey = new DataColumn()
{
   dtCustomers.Columns("CustomerID")
}

// Multiple columns PK
dtEmployees.PrimaryKey = new DataColumn()
{
   dtEmployees.Columns("LastName"),
   dtEmployees.Columns("FirstName")
}
```

В първия случай на свойството **PrimaryKey** на таблицата **dtCustomers** се задава като стойност масив, съдържащ колоната **CustomerID** на същата таблица.

Във втория случай на свойството **PrimaryKey** на таблицата **dtEmployees** се присвоява масив, съдържащ две колони – **LastName** и **FirstName** - от същата таблица.

ForeignKey и Unique ограничения

В колекцията **Constraints** могат да се поставят два вида ограничения върху таблицата – ограничения по външен ключ (**ForeignKeyConstraint**) и ограничения по уникалност (**UniqueConstraint**).

ForeignKeyConstraint ограничението задава колона или група колони в таблица, чиито стойности трябва да се срещат в определена колона (колони) в друга таблица, т.е. изпълнява функцията на външен ключ (foreign key) в базата от данни.

UniqueConstraint определя една или няколко колони, за които всеки ред от таблицата трябва да има уникални стойности.

Дефиниране на UniqueConstraint и ForeignKeyConstraint

Да предположим, че имаме **DataSet** обект с име **ds** и таблици **Products**, **Customers** и **Orders** в него. Следващият пример демонстрира създаване на **UniqueConstraint** ограничение:

```
ds.Tables["Products"].Constraints.Add(
   new UniqueConstraint("UC_ProductName",
      ds.Tables["Products"].Columns["ProductName"]));
```

Това става като към колекцията **Constraints** на **DataTable** обекта, представящ таблицата **Products** се добави **UniqueConstraint** обект. В конструктора на **UniqueConstraint** обекта се задава за кои колони се отнася ограничението.

Ето пример как се създава и **ForeignKeyConstraint** ограничение:

```
ForeignKeyConstraint custOrderFK =
   new ForeignKeyConstraint("CustOrderFK",
      ds.Tables["Customers"].Columns["CustomerID"],
      ds.Tables["Orders"].Columns["CustomerID"]);

custOrderFK.DeleteRule = Rule.None;
// Cannot delete a customer value
// that has associated existing orders.

ds.Tables["Orders"].Constraints.Add(custOrderFK);
```

При дефинирането на **ForeignKeyConstraint** първоначално се създава обект от този тип, като му се подава име на ограничението, колона-родител (в случая **CustomerID** в таблицата **Customers**) и колона-наследник (в случая **CustomerID** в таблицата **Orders**). След това на свойството **DeleteRule** на ограничението се задава стойност **None**, което означава, че от таблицата-родител не може да се изтрие ред, ако в таблицата-наследник има съответстващ на него родителски ред. Накрая ограничението се добавя към колекцията от ограничения на таблицата **Orders**.

Чрез свойствата **UpdateRule** и **DeleteRule** на **ForeignKeyConstraint** класа можем да задаваме какво да се случва с подчинените записи в таблицата-наследник при промяна на запис от таблицата-родител, от която те зависят. Можем да зададем каскадна промяна/изтриване, автоматично задаване на стойност **null**, автоматично задаване на подразбираща се

стойност, или както в примера – никакво действие, при което записите от таблицата-наследник не се променят, но се предизвиква изключение за нарушение на ограничението при опит за изтриване или невалидна промяна на реда от родителската таблица.

Потребителски изрази

ADO.NET позволява създаване на колони, базирани на изрази. Тези изрази могат да се използват за извършване на пресмятания върху стойности на колони в един ред или изчисляване на агрегатни функции върху множество редове.

Изразите, които се използват за дефиниране на колоната, се задават от програмиста и се присвояват на свойството **`Expression`** на класа **`DataColumn`**. Пример за такъв израз може да бъде следният:

```
Sum([UnitPrice] * [Quantity])
```

В колона, дефинирана чрез израз, реално не се съхранява никаква информация, а стойностите в нея се изчисляват при обръщение към тях. Когато се зададе свойството **`Expression`**, автоматично **`ReadOnly`** свойството за съответната колона става **`true`**.

Колоните, дефинирани чрез израз, могат да се използват съвместно с релациите родител/наследник, които ще бъдат обяснени по-подробно малко по-нататък. След дефиниране на релация, тя може да се използва за създаване на колона, чиито стойности се изчисляват на базата на стойности от свързаните таблици. В този случай могат да се използват и агрегатните функции **`Avg(…)`**, **`Count(…)`**, **`Sum(…)`**, **`Min(…)`** и **`Max(…)`**.

Потребителските изрази могат да се използват за създаване на следните видове колони:

- калкулирана колона;
- агрегирана колона;
- конкатенация на колони;
- обръщение към таблица родител или наследник (с използване на дефинирана релация).

Пример за дефиниране на колона чрез израз

Следващият пример показва как можем да дефинираме калкулирана колона за изчисляване на цена с включен данък добавена стойност (ДДС). В него се създава колона **`priceVat`**, задава се стойност **`Price * 1.2`** на свойството **`Expression`** на колоната и тя се добавя към колоните на таблицата **`productsTable`**. В случая стойностите в колоната ще бъдат изчислени чрез умножаване на стойностите в колоната **`Price`** по **`1.2`**:

```
// Calculated field – VAT
DataColumn priceVat = new DataColumn(
   "Price(VAT)", typeof(decimal));

priceVat.Expression = "Price * 1.2";

productsTable.Columns.Add(priceVat);
```

DataRelation обекти

DataRelation класът се използва за създаване на връзки между таблици предимно с навигационна цел. Те се използват често при работа с таблици в отношение родител-наследник (master-detail таблици). Чрез тях лесно се извлича списъкът на редове от подчинената таблица (таблицата-наследник), свързани с ред от главната таблица (таблицата-родител).

DataRelation обектите се съхраняват в колекцията **Relations** на **DataTable**.

Релации и ограничения

При създаването на **DataRelation** между две таблици не е задължително да се създадат и съответните ограничения (**ForeignKeyConstraint** и **UniqueConstraint**). По подразбиране се създава **UniqueConstraint** върху колоната от родителската таблица и **ForeignKeyConstraint** върху колоната от таблицата-наследник. Това позволява да се гарантира интегритетът на данните в таблиците, но ако такава функционалност не е нужна може да се изключи. За целта се използва конструктор на **DataRelation**, който приема като един от параметрите си булева стойност, указваща да се създадат ли ограничения или не.

DataRelation се създава по име на релацията и две референции към колони от две таблици – съответно родителя и наследника. Възможно е двете колони да принадлежат на една и съща таблица. Ето прост пример:

```
// Create the DataSet ds and add the tables Customers and Orders
DataSet ds = ...;

DataColumn parentColumn =
   ds.Tables["Customers"].Columns["CustID"];

DataColumn childColumn =
   ds.Tables["Orders"].Columns["CustID"];

// Create DataRelation
DataRelation relCustOrder = new DataRelation(
   "CustomersOrders", parentColumn, childColumn);
```

```
// Add the relation to the DataSet
ds.Relations.Add(relCustOrder);
```

В примера се създава релация между родителската таблица **Customers** и таблицата-наследник **Orders**.

Релации и потребителски интерфейс

Релациите родител/наследник често се използват за навигация в потребителския интерфейс. Чрез тях могат да се осъществява връзка и в двете посоки – от родителя към наследника и обратно. С тяхна помощ могат да се моделират връзки едно към много и много към много. Използват се за реализиране на master-detail връзки между таблици.

Например, може да се създадат два **DataGrid** контрола (DataGrid контролите визуализират таблични данни), единият от които изобразява родителска таблица, а другият – таблица-наследник. Чрез релация лесно може да се имплементира функционалност, която при избор на ред от родителската таблица представя в таблицата-наследник, редовете, свързани с избрания ред.

Релации и потребителски изрази

Релациите често се използват при дефиниране на колони с потребителски изрази. Те позволяват изразите да не се базират само на стойностите в един ред или една таблица, а на стойности в свързани таблици. Така в родителската таблица може да се създаде колона с информация, която се извлича от таблицата-наследник и обратно.

Релациите често се използват съвместно с агрегатни функции, за създаване на колони с агрегатни изрази. Обикновено се агрегира информацията от редовете в таблицата-наследник, съответстващи на ред от родителската таблица. Например в таблицата родител може да се създаде агрегирана колона, която показва колко са на брой съответните записи в таблицата-наследник.

Основни методи, използващи релации

За навигация между свързани с релация таблици се използват два метода на класа **DataRow**. Методът **GetChildRows()** връща масив от **DataRow** обекти от таблицата-наследник, които са наследници на реда, за който е извикан. Методът **GetParentRow()** се прилага към ред от таблицата-наследник и връща съответния ред от родителската таблицата.

Следващият пример илюстрира използването на релации:

```
dsNorthwind.Relations.Add("FK_CustomersOrders",
   dtCustomers.Columns["CustomerID"],
   dtOrders.Columns["CustomerID"], true);
```

```
// Process all customers and their orders
foreach (DataRow drCustomer in
   dsNorthwind.Tables["Customers"].Rows)
{
   foreach (DataRow drOrder in drCustomer.
      GetChildRows("FK_CustomerOrders"))
   {
      // Do something with the rows
   }
}
```

В примерния код най-напред в колекцията от релации на обекта **dsNorthwind** от клас **DataSet** се добавя релация, свързваща таблиците **dtCustomers** и **dtOrders** въз основа на колоните **CustomerID**. След това в цикъл за всеки ред от **dtCustomers** се обработват всички негови редове-наследници от **dtOrders**.

Класът DataView

Класът **DataView** наподобява изгледите (view обектите) в базата от данни. Той позволява да се представят само част от данните от определена таблица. Чрез него могат да се изграждат различни изгледи на едни и същи данни.

При създаване на **DataView** обект на конструктора на класа се подава референция към съществуваща таблица. Друг начин да се създаде изглед за дадена таблица в ADO.NET е като се използва свойството **DefaultView** на класа **DataTable**, което връща **DataView** обект към таблицата.

DataView се използва за изпълнение на две основни операции върху данните в една таблица:

- филтриране на редове;
- сортиране на редовете.

Филтрирането може да става по зададен израз или според състоянието на редовете. Сортирането става по колона.

Филтриране чрез израз

DataView може да се използва за създаване на динамичен изглед на данните, който се базира на израз за филтриране. Можем например да извлечем от една таблица с продажби всички, които са реализирани в определена държава. Този метод за филтриране прилича на филтриране чрез **WHERE** клауза в SQL заявка. Дори синтаксисът е подобен. Филтриращият израз се задава като стойност на свойството **RowFilter** на **DataView** обект. Подобно на условието в **WHERE** клаузата при SQL заявки, филтри-

ращият израз може да съдържа различни оператори за сравнение и функции. Ето пример за такъв израз:

```
ds.Tables["Customers"].DefaultView.RowFilter =
  "Country='Bulgaria' AND City='Sofia'";
```

Горният филтър ще създаде изглед, съдържащ всички клиенти от София.

Филтриране по версията на данните

DataView класът има свойство **RowStateFilter**, което позволява да филтрираме редовете според тяхното състояние. Това свойство приема стойности от изброения тип **DataViewRowState**, например **Added**, **Deleted**, **Unchanged**, **CurrentRows**, **ModifiedCurrent** и др. Те отразяват състоянието на реда в таблицата. Например, при запълване на таблица от **DataSet**, редовете в нея са в състояние **Unchanged**. Ако изтрием ред от таблицата, той остава в нея, но се маркира като **Deleted**. Ако добавим ред, той получава състояние **Added**.

Следва пример за филтриране по този признак. Създава се изглед, който съдържа променените и добавени редове в таблицата:

```
DataTable usersTable = ...;

DataView usersView = new DataView(usersTable);

// Show only modified versions of current rows and new rows
usersView.RowStateFilter =
  DataViewRowState.ModifiedCurrent |
  DataViewRowState.Added;
```

Сортиране по колона (колони)

За да се сортират данните в **DataView**, трябва да се зададе стойност на неговото свойство **Sort**, която е име на колона. Изразът за сортиране използва същия синтаксис като **ORDER BY** клаузата на SQL заявка.

Редът на сортиране се определя от типа на данните в колоната, по която се сортира.

Символните низове се сортират лексикографски, докато числата се сортират по големина. Може да се добави **ASC** или **DESC** към сортиращия израз, съответно за сортиране в нарастващ или намаляващ ред.

Възможно е сортиране по няколко колони. Ако на свойството **AplyDefaultSort** се зададе стойност **true**, редовете в таблицата се сортират в нарастващ ред по първичния ключ.

Следва пример за сортиране. В него редовете се сортират в нарастващ ред по стойността на колоната **Country**:

```
dsTables["Customers"].DefaultView.Sort = "Country ASC";
```

Запазване и зареждане на данните от DataSet

Една от важните характеристики на ADO.NET е неговата тясна обвързаност с XML. Както вече обяснихме, схемата на един **DataSet** може да се опише в **XSD** файл, но това не е всичко. ADO.NET позволява прочитане на данните от външен източник (файл, поток) в **DataSet** обект, както и съхраняване на съдържанието на **DataSet** в такъв източник. Използваният формат на данните във външния източник е XML. За целта се използват два основни метода на класа **DataSet**: **ReadXml(…)** и **WriteXml(…)**, които имат няколко версии, в зависимост от това с какъв източник се работи. Тези методи имат възможност за задаване и на някои допълнителни параметри, свързани например със схемата на данните. В тази точка ще се спрем по-подробно на тези два метода.

DataSet.ReadXml()

Съдържанието на един **DataSet** обект може да бъде взето от XML поток или документ. При това .NET Framework дава голяма гъвкавост относно това каква информация да се зареди от XML източника, както и как точно да се създаде схемата на **DataSet** обекта. За това се използва методът **ReadXml(…)** на класа **DataSet**. Той е предефиниран за различните източници на XML данните. Може да чете от следните източници:

- Отворен за четене поток. В този случай методът приема като параметър обект от тип **Stream** или **TextReader**.
- Символен низ. Този тип параметър означава път до файл или URL адрес. Ако файлът не съществува или потребителят няма право на достъп до него, методът предизвиква изключение.
- XML четец. В този случай методът приема като параметър обект от тип **XmlReader**.

Ето кратък пример за зареждане на **DataSet** обект от XML файл:

```
DataSet dsOrders = new DataSet();
dsOrders.ReadXml("orders.xml");
```

Режими на четене

Всеки от горните варианти на **ReadXml()** е предефиниран, така че да приема и един незадължителен параметър, който задава режим на четене. Този аргумент е от тип **XmlReadMode** и определя как ще се създаде схемата на **DataSet** обекта. Последното зависи и от това дали обектът вече има схема или няма. Текущата схема на данните може да се запази, да се прочете наново или да се генерира (извлече) от самите данни.

XmlReadMode е енумерация и има следните членове:

- **DiffGram** – прочита **DiffGram** съдържание и добавя данните към текущата схема (**DiffGram** е XML формат, който се използва за запазване на текущото и оригиналното съдържание на един **DataSet** обект). При това новите и съществуващите редове се сливат, ако стойностите на уникалните им идентификатори съвпадат.
- **Fragment** – чете XML фрагменти до достигане края на потока. Фрагментите, които имат същата схема като **DataSet** обекта, се добавят към съответните таблици. Останалите фрагменти се игнорират.
- **IgnoreSchema** – пренебрегва вградената схема и зарежда данните в **DataSet** обекта като използва неговата схема. Данните, които имат различна схема се пренебрегват. Ако **DataSet** обектът няма схема, не се зареждат данни.
- **InferSchema** – пренебрегва вградената в данните схема (ако има такава), извлича (генерира) схемата от самите данни и ги зарежда в **DataSet** обекта. Извлечената схема може да не описва точните типове на отделните колони в таблиците, защото такава информация не винаги може да бъде извлечена от самите данни. Ако обектът вече има схема, тя се разширява чрез добавяне на нови таблици и колони. При конфликт се предизвиква изключение.
- **ReadSchema** – прочита вградената в данните схема и зарежда **DataSet** обекта. Ако той има схема, могат да се добавят нови таблици, но се предизвиква изключение, ако таблица от вградената схема вече съществува в **DataSet** обекта.
- **Auto** – тази е стойността по подразбиране. Ако е зададена тя, се изследва XML съдържанието и ако то съдържа **DiffGram** (оригиналните данни заедно с извършените промени по тях), се използва **DiffGram**. Ако **DataSet** обектът има схема или документът има вградена схема, използва се **ReadSchema**. Ако **DataSet** обектът няма схема и XML съдържанието няма вградена схема, използва се **InferSchema**.

DataSet.WriteXml()

Съдържанието на **DataSet** обект може да се запише в XML формат, като това представяне може да съдържа или не схемата на обекта. Ако схемата е включена в XML съдържанието, тя се записва като стандартна **XSD** схема. При записване на данните в XML формат се записва текущата версия на редовете, но ако се записва като **DiffGram** се записва и оригиналната версия.

За записване на XML представянето на **DataSet** във файл, поток или XML четец се използва методът **WriteXml(…)**. Подобно на **ReadXml(…)**, **WriteXml(…)** е предефиниран за различните източници на данни, в които

може да се съхрани съдържанието. Той може да приема като параметър **Stream**, **XmlWriter**, **TextWriter** или **string**, като в последния случай параметърът указва път или URL, където да се запише съдържанието.

Режими на записване

Както и при **ReadXml(…)**, всеки вариант на **WriteXml(…)** е предефиниран да приема един незадължителен параметър, който задава режим на записване. Той е от тип **XmlWriteMode** и указва как да се запише съдържанието. Тази енумерация има следните членове:

- **DiffGram** – записва оригиналните данни и нанесените в тях промени (т.е., текущите стойности).
- **IgnoreSchema** – записва текущото съдържанието на **DataSet** обекта като XML данни, без да включва схемата. Това е стойността по подразбиране.
- **WriteSchema** – записва текущото съдържание на **DataSet** обекта, като записва и схемата като вградена XML Schema.

Синхронизация на DataSet с XmlDataDocument

DataSet класът предоставя релационно представяне на данните, докато средствата за работа с XML работят с йерархични данни. .NET Framework дава възможност едновременно да се работи и с двете представяния на едни и същи данни. Това се постига чрез синхронизация на **DataSet** и **XmlDataDocument**, който осигурява йерархичния изглед на данните. При синхронизация двата обекта използват едни и същи данни и когато те се променят през единия, промените се отразяват и в другия. Това дава голяма гъвкавост, тъй като позволява едно приложение да ползва едновременно средствата за работа с XML и обектния модел на ADO.NET за работа в несвързана среда (**DataSet**, **DataTable** и т. н.).

Различни начини да се извърши синхронизацията, както и предимствата на използването на **XmlDataDocument**, ще разгледаме малко по-нататък.

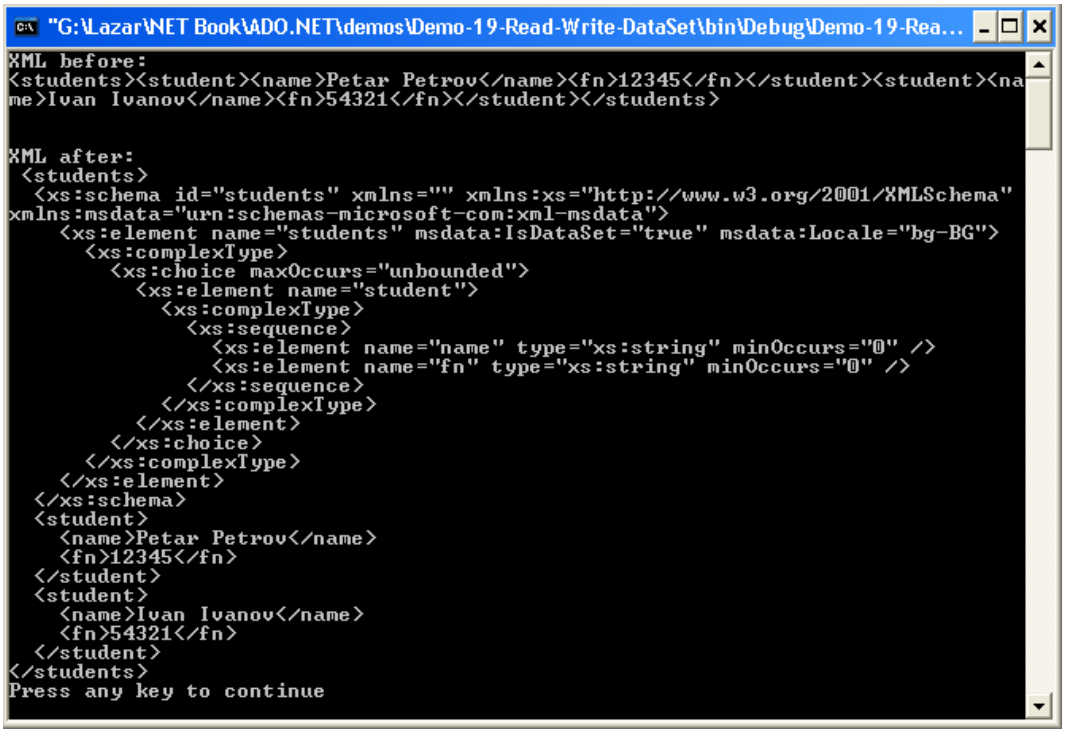
ReadXml() и WriteXml() – пример

Следващият пример илюстрира използването на двата метода, които дискутирахме – **ReadXml(…)** и **WriteXml(…)**:

```
// The DataSet that will be loaded from XML data
DataSet dsStudents = new DataSet();

// A string that contains the XML data
string xmlStudentData = "<students>" +
   "<student><name>Petar Petrov</name><fn>12345</fn>" +
   "</student><student><name>Ivan Ivanov</name>" +
   "<fn>54321</fn></student></students>";
```

```
// Create a StringReader used to pass the XML data
// from the string to the DataSet's ReadXml(...) method
StringReader srXmlStudents = new StringReader(xmlStudentData);
dsStudents.ReadXml(srXmlStudents, XmlReadMode.InferSchema);
srXmlStudents.Close();

// Create a StringWriter to store the data from the DataSet
StringWriter swStudents = new StringWriter();
dsStudents.WriteXml(swStudents, XmlWriteMode.WriteSchema);
string strStudents = swStudents.ToString();
swStudents.Close();

// Print to the console the XML before reading it in the
// DataSet, and the XML produced by the WriteXml(...) method
Console.WriteLine("XML before:\n" + xmlStudentData +
   "\n\n\nXML after:\n " + strStudents);
```

В резултат на изпълнение горният пример извежда следния изход:



Как работи примерът?

Първоначално създаваме **DataSet** обект **dsStudents**, в който ще прочетем XML съдържанието. В случая използваме XML, зададен в символния низ **xmlStudentData**. Той съдържа имената и факултетните номера на двама студенти.

След това създаваме четец **srXmlStudents** от тип **StringReader**, който обвива съдържанието на низа и се подава на метода **ReadXml(…)** на **dsStudents**. В случая използваме версията на метода, която приема поток и като незадължителен параметър задаваме **InferSchema**, тъй като **dsStudents** не съдържа схема, нито XML данните имат зададена такава. В резултат от изпълнението на метода информацията за двамата студенти се записва в **dsStudents** в таблица с име **student**. Тя има две колони – **name** и **fn**. Схемата се извлича от XML данните. ADO.NET приема, че двете колони са символни низове и са незадължителни. Дали това е реално така ADO.NET няма как да се сети (няма изкуствен интелект).

След това създаваме четец **swStudents** от тип **StringWriter**, в които записваме данните от **dsStudents** чрез обръщение към **WriteXml(…)**. Използваме версията, която приема поток и като незадължителен параметър задаваме **WriteSchema**. Накрая извличаме XML съдържанието от четеца в низ и отпечатваме в конзолата първоначалния XML и извлечения от **dsStudents**. От резултата се вижда, че в XML съдържанието, извлечено от **dsStudents**, е записана и схемата като XSD. Това се дължи на използването на параметъра **WriteSchema**.

Използване на DataAdapter

Както вече обяснихме, **DataSet** обектите нямат постоянна връзка с източника на данни (базата данни). Логично е да съществува някакъв автоматичен начин за извличане на данни от източника в **DataSet** обекта и за обновяване след това на данните в източника, след като тяхното копие в **DataSet** обекта е било променено. Именно за тази цел се използва класът **DataAdapter**. Той осъществява връзката между **DataSet** обекта и източника на данни.

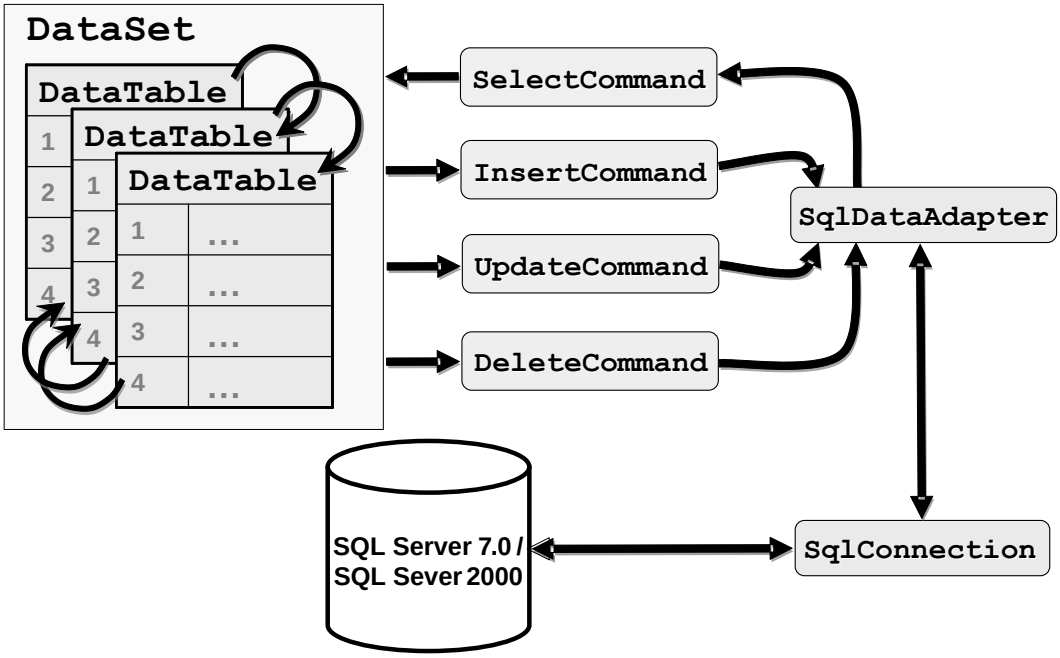
При работа с източник, който е релационна база от данни, се използват наследници на класа **DataAdapter**, които наследяват **DbDataAdapter** и имплементират интерфейса **IDBDataAdapter**, например **SqlDataAdapter** за MS SQL Server, **OleDbDataAdapter** за OLE DB и т.н.

Архитектура на класа DataAdapter

DataAdapter дефинира команди, чрез които извършва извличането на данните от източника и тяхното обновяване. По-конкретно, за извличане на запис се използва командата **SelectCommand**, за добавяне на запис – **InsertCommand**, за промяна на запис – **UpdateCommand** и за изтриване на запис – **DeleteCommand**. Тези команди представляват SQL заявки, които извършват съответните действия. Ще се спрем подробно на тях малко по-нататък.

За да работи правилно един **DataAdapter**, освен командите, трябва да му се зададе и връзка към базата данни (**Connection**), през която да ги изпълнява.

На фигурата е показана архитектурата на **DataAdapter**:



Адаптерни класове за различните доставчици

DataAdapter е абстрактен клас, предоставящ базова функционалност за имплементиране на несвързан достъп до данни. На практика в приложенията се използват наследници на **DataAdapter** класа, които са специфични за различните доставчици на данни. Такива са **SqlDataAdapter**, **OleDbDataAdapter** и **OdbcDataAdapter**, които се намират в съответните пространства от имена **System.Data.SqlClient**, **System.Data.OleDb** и **System.Data.Odbc**. За реализиране на извличането на данни в **DataSet** **DataAdapter** използва вътрешно **DataReader**, но това е подробност на имплементацията.

Създаване на DataAdapter

Има няколко начина за създаване на **DataAdapter** обект. Конструкторът е предефиниран така, че да приема различни параметри. Освен безаргументния конструктор, има още три версии на конструктора.

Възможно е обектът да се създаде по низ, съдържащ **SELECT** заявка и низ за връзка. Друг вариант е на конструктора да се подадат низ, съдържащ **SELECT** заявка и обект за връзка към базата. Следващият пример демонстрира създаване на адаптер по този начин:

```
string strCon = "DataSource=(local);IntegratedSecurity=SSPI;" +
  "InitialCatalog=Northwind";
```

```
SqlConnection cnNorthwind = new SqlConnection(strCon);

string strSelect = "SELECT * FROM Orders";

SqlDataAdapter daOrders = new SqlDataAdapter(
  strSelect, cnNorthwind);
```

Горните начини на създаване на **DataAdapter** обаче не са особено удобни, ако искаме да използваме параметризирана заявка или съхранена процедура за извличане на данните. В тези случаи се създава първо **Command** обект и той се подава на конструктора на адаптера. Следващият пример демонстрира този подход:

```
string strCon = "DataSource=(local);IntegratedSecurity=SSPI;" +
  "InitialCatalog=Northwind";
SqlConnection cnNorthwind = new SqlConnection(strCon);

string strSelect = "MyStoredProcedure";

SqlCommand cmdSelect = new SqlCommand(strSelect, cnNorthwind);
cmdSelect.CommandType = CommandType.StoredProcedure;

SqlDataAdapter daOrders = new SqlDataAdapter(cmdSelect);
```

Класът **DataAdapter** има свойство **SelectCommand**, което съдържа обект от тип **Command**, който или се създава неявно при първите два варианта, или сочи подадения **Command** обект при третия вариант. Този обект съдържа заявката за извличане (или името на съхранената процедура). Вътрешно **DataAdapter** използва тази заявка за да извлича данните, като за целта отваря **DataReader**.

Методът Fill() на класа DataAdapter

Методът **Fill(…)** се използва за извличане на данни от източника на данни. Чрез него може да се запълни **DataSet** обект или самостоятелен **DataTable** обект. При извикване на този метод се изпълнява заявката, която се съдържа в свойството **SelectCommand** на **DataAdapter** обекта и данните се извличат в таблица от **DataSet** обекта или в **DataTable** обекта. Същевременно се извлича и информация за схемата на таблицата. Тази информация обаче е много ограничена – съдържа единствено имената и типовете на колоните в таблицата.

Нека се върнем отново на първия от горните два примера. С така създадения **DataAdapter** извличането на данните от таблица **Orders** от базата **Northwind** и записването им в таблица **Orders** в **DataSet** обект **dsNorthwind** може стане по следния начин:

```
DataSet dsNorthwind = new DataSet();
```

```
daOrders.Fill(dsNorthwind, "Orders");
```

По подобен начин става запълването на самостоятелна таблица **DataTable** (адаптерът е създаден по същия начин):

```
DataTable dtOrders = new DataTable("Orders");
daOrders.Fill(tableOrders);
```

DataAdapter сам се грижи за връзката с източника – не е нужно ние явно да я отваряме и затваряме. Ако връзката е затворена, той я отваря и като приключи работа, я затваря. Ако връзката е била отворена, тя остава пак отворена след приключване на работа. Можем да обобщим: след приключване на работата си **DataAdapter** оставя връзката в състоянието, в което я е получил.

Свойството MissingSchemaAction

Класът **DataAdapter** има свойство **MissingSchemaAction**, което указва какво да се случи ако съществуващата схема на **DataSet** обекта не съответства на извличаната схема. Това свойство приема стойности от изброения тип **MissingSchemaAction**, който включва следните членове:

- **Add** – липсващите колони се добавят, така че да се допълни схемата на **DataSet** обекта. Това е стойността по подразбиране.
- **AddWithKey** – добавят се липсващите колони и информацията за първичните ключове.
- **Ignore** – допълнителните колони се игнорират.
- **Error** – генерира се изключение, ако зададеното съответствие на колони липсва.

Запълване на няколко таблици

Има няколко начина да запълним повече от една таблица в един **DataSet** обект:

- Извикваме няколко пъти метода **Fill(…)** на адаптера, като при всяко извикване променяме **SELECT** заявката на **SelectCommand** свойството на адаптера, така че да се извличат данните от различни таблици.

При неколкократно извикване на метода Fill(…) отваряйте връзката преди серията извиквания и след това я затваряйте. В противен случай за всяко извикване на Fill(…) адаптерът ще отваря и затваря връзката, което е тежка операция.

- Използваме няколко различни **DataAdapter** обекта, всеки от които извлича данните от различна таблица.
- Използваме пакетна заявка (batch query) или съхранена процедура, които връщат последователно съдържанието на няколко таблици. При пакетната заявка в свойството **SelectCommand.CommandText** на адаптера се задават няколко **SELECT** заявки, които се отделят с точка и запетая (;).

В последния случай адаптерът автоматично създава таблиците в **DataSet** обекта и им дава имена **Table**, **Table1** и т.н., ако не им зададем явно име. Ако зададем някакво име, например **Orders**, адаптерът ще създаде таблици **Orders**, **Orders1** и т.н. След запълване можем да преименуваме таблиците както искаме.

Задаване на съответствие за таблици и колони

DataAdapter класът има свойство **TableMappings**, което представлява колекция от **DataTableMapping** обекти. Чрез него могат да се задават съответствия между имена на таблици в източника на данни и в **DataSet** обекта. Обикновено се използва при извличане едновременно на много таблици от източника, тъй като тогава на таблиците се задават имена по подразбиране. Ако зададем съответствия, когато се създава всяка от таблиците в **DataSet** обекта, на нея ще се даде името, което сме посочили в **DataTableMapping** обекта. Ето как може да стане това:

```
daOrders.TableMappings.Add("Orders", "MyOrders");
```

В резултат от горния ред код, когато се извлече съдържанието на таблица **Orders** от източника и се създаде съответна на нея таблица в **DataSet** обекта, последната ще има име **MyOrders**.

По подразбиране, когато се запълва таблица в **DataSet** обект, се използват същите имена на колони като в таблицата в източника. Ако искаме да зададем други имена на колоните, можем да използваме свойството **ColumnMappings**, което имат обектите в **TableMappings** колекцията на адаптера. Ето един кратък пример:

```
daOrders.TableMappings["Orders"].ColumnMappings.
  Add("id", "OrderID");
```

В примера се указва на адаптера, че колоната **id** от таблицата **Orders** от базата данни трябва да приеме името **OrderID** в съответната таблица от **DataSet** обекта.

Ако имената на извлечените от източника таблици и колони не съответстват на тези в **DataSet** обекта, свойството **MissingMappingsAction** определя какви действия да се предприемат. По подразбиране се създават

липсващите обекти. Другите възможности са те да бъдат пренебрегнати или да се предизвика изключение.

Извличане на информация за схемата на източника

Схемата на източника на данни може да се извлече чрез метода **FillSchema(…)** на класа **DataAdapter**. Той извлича информация за схемата, свързана с резултата от заявката в **SelectCommand** свойството на адаптера. Методът създава таблица в **DataSet** обекта със съответните колони и конфигурира свойствата им според източника. Когато конфигурира **AutoIncrement** свойството, не задава стойности за **AutoIncrementSeed** и **AutoIncrementStep**. **FillSchema(…)** конфигурира първичния ключ и ограниченията по уникалност за таблиците в **DataSet** обекта.

Методът приема и допълнителен аргумент, освен **DataSet** обекта, който задава дали извлечената схема да се промени в съответствие със зададените за адаптера съответствия на колоните и таблиците.

Ако **FillSchema(…)** се използва за таблица с дефинирана схема, тя не се заменя с извлечената, а се допълва с нови колони, ако в извлечената схема има такива.

Когато се използва заявка, връщаща много таблици, **FillSchema(…)** извлича схемата само на първата върната таблица. Ако искаме информация за схемите на всички таблици, трябва да използваме **Fill(…)** метода със стойност **AddWithKey** на аргумента **MissingSchemaAction**.

Свойства AcceptChangesDuringFill и ContinueUpdateOnError

Свойството **AcceptChangesDuringFill** на **DataAdapter** определя при добавяне на редове в таблица от **DataSet** обект каква ще бъде стойността на **RowState** за тези редове. Ако **AcceptChangesDuringFill** е **true**, след добавяне редовете ще имат за **RowState** стойност **Unchanged**. Ако на свойството е зададена стойност **false**, **RowState** на въведените редове ще бъде **Added**.

Стойността на свойството **ContinueUpdateOnError** се използва при обновяване на данните в източника. Ако е **true**, когато възникне грешка при обновяване на ред не се предизвиква изключение, а се продължава с обновяване на следващите редове. В случай, че стойността на свойството е **false**, ако настъпи грешка при обновяване на ред се предизвиква изключение. Обработката на изключението определя дали обновяването се продължава за следващите редове.

Обикновено за **ContinueUpdateOnError** се задава **false** в случай, че промените върху **DataSet** обекта са част от транзакция. Тогава обновяването трябва да успее или за всички редове, или да не се извърши за нито един.

Ако възникне грешка, обработчикът на изключението отменя транзакцията.

Събития на DataAdapter

Събитието **FillError** възниква, когато въвежданите данни нарушават някое ограничение в **DataSet** обекта или не може да се извърши конверсия без загуба на точност между тип в източника и .NET Framework тип. При настъпване на събитието текущият ред не се добавя към таблицата в **DataSet** обекта. Обработчикът на събитието може да поправи грешката и редът да бъде добавен към таблицата или да укаже да бъде игнориран.

DataAdapter предлага и две събития, които възникват при обновяване на данните в източника. Събитието **RowUpdating** възниква преди направените промени в реда в **DataSet** да бъдат пренесени в източника. Обработчикът му може да извърши някаква допълнителна обработка върху реда или да не го обнови в източника. **RowUpdated** събитието възниква след като ред е обновен в източника и се използва за извършване на обработка, свързана с грешки, възникнали при обновяването. С негова помощ се разрешават конфликтите при обновяване на данни в базата данни.

Обновяване на данните в източника

Тъй като **DataSet** не е свързан директно с източника на данни, при промяна на извлечените данни промените не се внасят автоматично и в източника. За да извършим обновяването, трябва да използваме метода **Update(…)** на класа **DataAdapter**. При неговото изпълнение адаптерът проверява таблиците в **DataSet** обекта за добавени, изтрити и променени редове и внася промените в таблиците в съответните им таблици в източника. За всеки от засегнатите редове той изпълнява съответно **INSERT**, **DELETE** или **UPDATE** заявка. В следващата точка ще разгледаме откъде се взимат тези заявки.

Свойства InsertCommand, DeleteCommand и UpdateCommand

Освен свойството **SelectCommand**, което вече разгледахме, класът **DataAdapter** дефинира още свойствата **InsertCommand**, **DeleteCommand** и **UpdateCommand**. Те представляват **Command** обекти, съдържащи заявките, които се изпълняват съответно при добавяне, изтриване и промяна на редове в процеса на обновяване на данните в източника. На тези свойства трябва да се зададат подходящи SQL команди преди извикване на метода **Update(…)** за обновяване на базата данни. Това може да се направи по два начина – автоматично (чрез класа **CommandBuilder**) или ръчно (чрез потребителска логика).

Класът CommandBuilder

Единият начин да зададем стойности на **InsertCommand**, **DeleteCommand** и **UpdateCommand** е като използваме обект от клас **CommandBuilder**. Той генерира автоматично обектите, които се съдържат в горните свойства. За целта **CommandBuilder** използва свойството **SelectCommand**, за да получи информацията, необходима за създаване на съответните командни обекти за обновяване.

Генерираните обекти за обновяване съдържат параметризирани SQL заявки. Те използват както оригиналните стойности на данните, така и текущите (специално в случая на промяна). Например, за промяна на запис в източника, записът бива намиран по оригиналните стойности и след това полетата му биват променяни с текущите им стойности.

CommandBuilder – пример

Ето един пример за използване на **SqlCommandBuilder**:

```
SqlConnection cn = new SqlConnection(...);

// Create data adapter for the table Users
string strSelect = "SELECT * FROM Users";
SqlDataAdapter da = new SqlDataAdapter(strSelect, cn);

// Create command builder for the adapter
SqlCommandBuilder cb = new SqlCommandBuilder(da);

// Retrieve the Users table from the database
DataSet ds = new DataSet();
da.Fill(ds);

// Make some changes in the DataSet
DataTable usersTable = ds.Tables[0];
usersTable.Rows[0]["username"] = "pesho";

// Apply the changes to the database
da.Update(ds);
```

В примера се създава адаптер по SQL командата "**SELECT * FROM Users**". След това се създава **SqlCommandBuilder** за този адаптер, който се грижи да генерира автоматично SQL командите за обновяване на базата данни, когато бъде извикан **Update(…)** методът на адаптера. След това от базата данни се извлича таблицата **Users** и се записва в **DataSet** обект. Следва промяна на първия ред от извлечената локално таблица и обновяване на базата данни. Благодарение на **CommandBuilder** обекта, свързан с адаптера, обновяването е успешно.

Как работи CommandBuilder?

При създаването на **CommandBuilder** обект той се свързва със събитието **RowUpdating** на адаптера и при изпълнение на **Update(…)** генерира динамично необходимите заявки за обновяване на базата данни.

Заявките се генерират в момента, в който потрябват за първи път (on demand). В горния пример се генерира само **UPDATE** заявка и то в момента, в който се направи опит за обновяване на променения запис от таблицата **Users**. Понеже в примера записи не се добавят и не се изтриват, **CommandBuilder** изобщо не генерира **INSERT** и **DELETE** команди.

За генерирането на заявките **CommandBuilder** първо изпълнява **SELECT** командата от адаптера в режим, в който извлича от нея само метаданните на върнатия резултат (имената и типовете на колоните) без самите данни. След това по извлечевните метаданни той построява подходящи **INSERT**, **UPDATE** и **DELETE** параметрични заявки като използва информацията за първичния ключ и уникалните колони от метаданните за таблицата.

Ограничения на CommandBuilder

Създаването на команди за обновяване с помощта на **CommandBuilder** е лесно, но има известни ограничения. Заявката в **SelectCommand** трябва да връща резултати само от една таблица, в противен случай се хвърля изключение. Друго ограничение е изискването заявката да включва първичния ключ или поне една колона с уникални стойности, за да могат генерираните **UpdateCommand** и **DeleteCommand** да определят еднозначно редовете за промяна и изтриване, съответно. В предходния пример таблицата **Users** има дефиниран първичен ключ (колоната **id**):

Design Table 'Users'

	Column Name	Data Type	Length	Allow Nulls
🔑	id	int	4	
	username	varchar	20	
	passwd	varchar	20	✓

Още едно ограничение на **CommandBuilder** е, че за да работи правилно, генерираната команда **InsertCommand**, заявката в **SelectCommand** трябва да съдържа всички колони, за които няма стойности по подразбиране.

Използването на **CommandBuilder** е лесно и позволява генериране на команди за обновяване с много малко код, но недостатъците му (гореописаните ограничения, както и това, че не поддържа съхранени процедури) го правят неудобен в случаи, когато е нужна по-сложна логика за обновяване.

CommandBuilder, подобно на **DataAdapter**, има специфични имплементации за различните доставчици на данни.

Потребителска логика за обновяване на източника

Използването на **CommandBuilder** е удобно, но не винаги върши работа. Ето някои случаи, в които не може да се използва:

- ако искаме да извлечем данни чрез съединение на таблици (както вече споменахме, **CommandBuilder** може да използваме само ако извличаме данни от единствена таблица);
- при използване на съхранени процедури при обновяване (а те се използват често, тъй като предлагат по-голяма сигурност, по-лесна поддръжка, както и по-висока ефективност);
- ако е необходима по-сложна логика на управление на конкурентния достъп.

Ако не искаме да ползваме **CommandBuilder** за генериране на командите за обновяване, се налага сами да ги напишем (ще дадем пример малко по-нататък). Те съдържат параметризирани заявки. Когато добавяме параметрите в командата, трябва да зададем на коя колона от таблицата съответства всеки параметър. Можем да задаваме и коя версия на стойността да се използва в заявката – оригиналната или текущата. Освен параметризирани заявки може да използваме и съхранени процедури.

Извличане на обновени стойности в DataSet

Чрез **InsertCommand**, **DeleteCommand** и **UpdateCommand** можем да внесем в източника промените, направени в **DataSet** обекта, но обновените данни не се връщат автоматично в **DataSet** обекта. Обикновено това не е нужно – би трябвало след обновяването данните да имат едни и същи стойности в източника и **DataSet** обекта. Понякога, обаче, това не е така.

Например, ако имаме **AutoIncrement** колона, след обновяване в източника тази колона ще има различни стойности от тези в **DataSet** обекта (за редовете, които са добавени с **InsertCommand**). Тези стойности се генерират автоматично от сървъра за всеки запис в източника и не се взимат от **DataSet** обекта. Тъй като такива колони обикновено се използват като първичен ключ, се налага да се извлекат обратно в **DataSet** обекта генерираните в източника стойности за съответната колона.

Има три начина да се извлекат такива стойности след обновяване:

- След текста на съответната заявка за обновяване можем да добавим **SELECT** заявка, която извлеча променените стойности и ги записва в съответните параметри. В този случай стойностите се съдържат в първия ред, върнат от заявката, която обновява ред в източника.
- Можем да използваме за обновяване съхранена процедура и параметър, в който да се връща нужната стойност.
- Можем да извлечем новата стойност в обработчика на събитието **RowUpdated** на **DataAdapter**.

В следващия пример ще илюстрираме първия начин.

DataAdapter – пример

В следващия пример ще покажем използването на **`DataAdapter`** за запълване на **`DataSet`** и обновяване на данните в източника. Използван е SQL Server доставчик на данни. Ще използваме ръчно генерирани команди за извличане и обновяване на данните.

```
static void Main()
{
  string strCon = "Data Source=(local);" +
    "Integrated Security=SSPI;Database=Northwind";
  SqlConnection cnNorthwind = new SqlConnection(strCon);

  SqlCommand cmdSelect = CreateSelectCommand(cnNorthwind);
  SqlDataAdapter daEmployees = new SqlDataAdapter(cmdSelect);
  DataSet dsNorthwind = new DataSet();
  daEmployees.Fill(dsNorthwind, "Employees");

  // Set the AutoIncrement property of EmployeeID column
  DataTable employeesTable = dsNorthwind.Tables["Employees"];
  DataColumn columnEmployeeId =
    employeesTable.Columns["EmployeeID"];
  columnEmployeeId.AutoIncrement = true;
  columnEmployeeId.AutoIncrementSeed = -1;
  columnEmployeeId.AutoIncrementStep = -1;

  // Create the commands for the data adapter
  daEmployees.InsertCommand = CreateInsertCommand(cnNorthwind);
  daEmployees.DeleteCommand = CreateDeleteCommand(cnNorthwind);
  daEmployees.UpdateCommand = CreateUpdateCommand(cnNorthwind);

  // Add new record and update the database
  DataRow row = employeesTable.NewRow();
  row["LastName"] = "Ivanov";
  row["FirstName"] = "Ivan";
  employeesTable.Rows.Add(row);
  daEmployees.Update(dsNorthwind, "Employees");
  Console.WriteLine("Inserted row id={0}.", row["EmployeeID"]);

  // Change the added record and update the database
  row["LastName"] = "Petrov";
  daEmployees.Update(dsNorthwind, "Employees");
  Console.WriteLine("Updated the row.");

  // Delete the added record and update the database
  row.Delete();
  daEmployees.Update(dsNorthwind, "Employees");
  Console.WriteLine("Deleted the row.");
```

```
}

static SqlCommand CreateSelectCommand(SqlConnection aConnection)
{
   string strSelect = "SELECT EmployeeID, LastName, FirstName " +
      "FROM Employees";
   SqlCommand cmdSelect = new SqlCommand(strSelect, aConnection);
   return cmdSelect;
}

static SqlCommand CreateInsertCommand(SqlConnection aConnection)
{
   string strInsert = "INSERT Employees(LastName, FirstName) " +
      "VALUES(@LastName, @FirstName);" +
      "SET @EmployeeID=Scope_Identity()";

   SqlCommand cmdInsert = new SqlCommand(strInsert, aConnection);
   SqlParameterCollection cparams = cmdInsert.Parameters;
   SqlParameter empID = cparams.Add("@EmployeeID", SqlDbType.Int,
      0, "EmployeeID");
   empID.Direction = ParameterDirection.Output;
   cparams.Add("@LastName", SqlDbType.NVarChar, 20, "LastName");
   cparams.Add("@FirstName", SqlDbType.NVarChar, 10,"FirstName");

   return cmdInsert;
}

static SqlCommand CreateUpdateCommand(SqlConnection aConnection)
{
   string strUpdate = "UPDATE Employees SET " +
      "LastName=@LastName, FirstName=@FirstName " +
      "WHERE EmployeeID=@EmployeeID";

   SqlCommand cmdUpdate = new SqlCommand(strUpdate, aConnection);
   SqlParameterCollection cparams = cmdUpdate.Parameters;
   SqlParameter empID = cparams.Add("@EmployeeID", SqlDbType.Int,
      0, "EmployeeID");
   empID.SourceVersion = DataRowVersion.Original;
   cparams.Add("@LastName", SqlDbType.NVarChar, 20, "LastName");
   cparams.Add("@FirstName", SqlDbType.NVarChar, 10,"FirstName");

   return cmdUpdate;
}

static SqlCommand CreateDeleteCommand(SqlConnection aConnection)
{
   string strDelete = "DELETE FROM Employees " +
      "WHERE EmployeeID = @EmployeeID";

   SqlCommand cmdDelete = new SqlCommand(strDelete, aConnection);
```

```
    SqlParameter empID = cmdDelete.Parameters.Add("@EmployeeID",
      SqlDbType.Int, 0, "EmployeeID");
    empID.SourceVersion = DataRowVersion.Original;

    return cmdDelete;
}
```

Как работи примерът?

В горния пример се извличат данни от таблицата **Employees** от стандартната за SQL Server база данни **Northwind**, съхраняват се в **DataSet** обекта **dsNorthwind** и върху тях последователно се извършват промени, които се нанасят в таблицата от базата.

За извличане на данните се използва методът **Fill(…)** на адаптера **daEmployees**. Той се създава чрез версията на конструктора, приемащ **Command** обект. След извличане на данните от базата и запазването им в таблица **Employees** в **dsNorthwind**, се конфигурира свойството **AutoIncrement** на колоната **EmployeeID**. Тази колона е първичен ключ на таблицата. Добра практика е автоматично генерираните стойности за тази колона в добавяните редове да започнат от отрицателна стойност и да намаляват. Това позволява лесно разграничаване на новите редове от старите, извлечени от базата данни.

След това се извършват промени върху информацията в **dsNorthwind** – добавя се ред, обновява се базата данни, променя се ред, отново се обновява базата данни и накрая се изтрива преди това добавеният ред.

За да работи правилно обновяването на базата данни преди да извършим промените създаваме подходящи параметрични команди за свойствата **InsertCommand**, **DeleteCommand** и **UpdateCommand** на **daEmployees**. За целта сме използвали различни функции – съответно **CreateInsertCommand(…)**, **CreateDeleteCommand(…)** и **CreateUpdateCommand(…)**. Те приемат като параметър обект за връзка с източника и връщат **Command** обект.

При създаването на **SelectCommand** селектираме всички колони от таблицата **Employees**, с които ще работим, и първичния ключ. Извличането на първичния ключ е важно, защото без него няма да можем да нанасяме промените в таблицата, понеже не можем да знаем за кой точно запис от базата данни се отнася текущият ред, който обновяваме.

При създаване на **InsertCommand** след текста на **INSERT** заявката сме добавили и **SET** команда. С нея извличаме автоматично генерираната от сървъра стойност на първичния ключ за последния добавен ред. За целта използваме функцията на SQL Server **Scope_Identity()**. При добавяне на параметрите за командата, съдържаща тази заявка, указваме, че параметърът **@EmployeeID** е изходен (**Output**) параметър, тъй като той ще съдържа върнатата генерирана стойност.

При създаване на **UpdateCommand**, указваме параметърът **@EmployeeID** да използва оригиналната версия на стойността в съответната колона за реда, който обновяваме. Така сме сигурни, че ще се използва стойността, която е заредена от базата, а не текущата стойност, която може случайно да е променена. Това е важно, защото по тази стойност заявката ще намери реда за обновяване в базата. (В случая, обаче, тази стъпка може да се пропусне, тъй като предполагаме, че стойността в тази колона няма да се променя от програмата).

Създаването на **DeleteCommand** работи аналогично на **UpdateCommand**.

Обновяване на свързани таблици

Когато извършваме обновяване на таблици, между които има връзки от тип родител-наследник (master-detail relationships), трябва да спазим определен ред на нанасяне на промените, за да избегнем нарушаване на целостта на данните. Ето една препоръчителна последователност от стъпки, която минимизира проблемите при обновяване на таблици във връзка родител-наследник:

1. Изтриваме премахнатите редове от таблицата-наследник.
2. Изтриваме премахнатите редове от родителската таблица.
3. Обновяваме променените редове в родителската таблица.
4. Добавяме новите редове в родителската таблица.
5. Обновяваме променените редове в таблицата-наследник.
6. Добавяме новите редове в таблицата-наследник.

Ако следваме горната последователност от стъпки, няма да бъдат нарушени евентуални ограничения (constraints) по таблиците от базата данни. Това не изключва, обаче, възможността да възникнат конфликти заради преждевременни промени, нанесени от други, паралелно работещи, потребители.

DataSet.GetChanges() и DataSet.HasChanges()

Методите **DataSet.GetChanges()** и **DataSet.HasChanges()** служат за извличане на информация за промените, направени в даден **DataSet**. Нека ги разгледаме в по-големи детайли.

Параметри на методите

Двата метода **GetChanges(…)** и **HasChanges(…)** имат безаргументна версия и версия, която приема като аргумент стойност от изброения тип **DataRowState**, показващ версията на данните.

Допустимите стойности на **DataRowState** следните:

- **Added** – добавени редове.

- **Deleted** – изтрити редове.
- **Detached** – редове, които са създадени, но не са добавени към нито една таблица.
- **Modified** – променени редове.
- **Unchanged** – редове, в които няма промяна от зареждането им в **DataSet** обекта или от последното извикване на **AcceptChanges()**.

Какво правят двата метода?

Методът **HasChanges(…)**, използван без аргумент, връща булева стойност, указваща дали в данните в **DataSet** обекта са извършвани някакви промени, т.е. дали са добавяни, изтривани или променяни редове. Ако му се подаде някой от горните параметри, върната стойност ще покаже дали са направени промени от посочения тип. Например, ако му подадем аргумент **DataRowState.Added**, методът ще върне стойност, указваща дали в **DataSet** обекта има добавени редове.

Методът **GetChanges(…)**, използван без аргумент, връща копие на **DataSet** обекта, което съдържа редовете, които са били променени по някакъв начин след зареждането на данните, или след последното извикване на **AcceptChanges()**. Ако му се подаде някой от описаните по-горе параметри, върнатият **DataSet** ще съдържа само редовете със съответната промяна. Например, ако му подадем **DataRowState.Added**, този метод ще върне копие на **DataSet** обекта, съдържащо само добавените редове.

Методът **AcceptChanges()** се използва за потвърждаване на промените, извършени в **DataSet** обекта. При него състоянието на редовете с версия **Added** и **Modified** става **Unchanged**, а редовете, маркирани като **Deleted** действително се изтриват от **DataSet** обекта. Извикването на този метод не се отразява по никакъв начин на източника на данни. Този метод се вика неявно при обръщение към **Update(…)** метода на класа **DataAdapter**.

Кога да използваме GetChanges() и HasChanges()?

Обикновено **GetChanges()** се използва, за да се извлекат от **DataSet** обекта само променените данни. Това е удобно, ако се предават данни от една машина на друга. В този случай има значение за производителността дали ще се предаде целият **DataSet**, или част от него.

Друг случай на използване на **GetChanges()** е обновяването на свързани таблици, което разгледахме накратко преди малко.

Обикновено първо се прави обръщение към метода **HasChanges(…)**, а след това, в случай че в **DataSet** обекта има промени, се вика **GetChanges()**.

DataSet.GetChanges() – пример

Следващият пример илюстрира използването на **HasChanges(…)** и **GetChanges(…)**:

```
if(! myDataSet.HasChanges(DataRowState.Modified))
{
  return;
}

// GetChanges for modified rows only
DataSet modifiedDataSet =
  myDataSet.GetChanges(DataRowState.Modified);

// Check the DataSet for errors
if (modifiedDataSet.HasErrors) {
  // Insert code to resolve errors
}

// After fixing the errors, update the data source
// with the DataAdapter used to fill the DataSet
adapter.Update(modifiedDataSet);
```

В примера с **HasChanges(…)** се проверява дали в **myDataSet** има променени редове (като на метода се подава аргумент **DataRowState.Modified**), и ако има, се създава копие на **myDataSet** чрез метода **GetChanges(…)**, което съдържа само променените редове. Проверява се дали в тях има грешки и след евентуално отстраняване на грешките промените се нанасят в източника на данни.

Грешките в DataSet и DataTable обектите

Грешките в **DataSet** обектите се задават от програмиста и могат да се извличат след това. Грешки могат да се задават за всяка таблица, за всеки ред и за всяка колона от всеки ред и могат да съдържат текстово описание. Най-често се използват за запазване на проблеми, възникнали при нанасянето на промени от **DataSet** в базата данни.

Основните методи и свойства за работа с грешки са следните:

- **DataSet.HasErrors** – връща **true** ако в някоя от таблиците в **DataSet** обекта има грешки.
- **DataTable.HasErrors** – връща **true** в таблицата има грешки.
- **DataRow.SetErrorOccured(column, errorMessage)** – задава грешка за дадена колона от даден ред.
- **DataTable.GetErrors()** – връща масив от всички редове от дадена таблица, които съдържат грешки.

- **DataRow.GetColumnsInError()** – връща масив от всички колони от даден ред, които съдържат грешки.
- **DataRow.GetColumnError(column)** – връща текстовото описание на грешката за дадена колона от даден ред.

Използването на грешки в таблиците ще демонстрираме малко по-нататък в примерите.

Несвързан модел – типичен сценарий на работа

В тази точка ще разгледаме последователността от основните операции, които се изпълняват при използване на несвързания модел за работа с данни. В някои случаи към описаните стъпки могат да се добавят нови или да се променят някои от тях. Ето описание на препоръчителни стъпки:

1. Зареждаме данните в **DataSet** обект от източник на данни. За целта можем да използваме метода **Fill(…)** на **DataAdapter**:

```
userDataAdapter.Fill(dsUsers);
```

   Възможно е данните да се заредят в **DataSet** обекта и по друг начин – например от XML файл, чрез метода **ReadXml(…)**.

2. След като сме заредили данните в **DataSet** обекта, извършваме някаква обработка върху тях – обикновено тя се състои в изтриване, добавяне и модификация на редове от таблиците.

3. Преди да внесем направените промени в източника, извличаме променените редове. Използваме метода **DataSet.GetChanges()**. Ако няма промени, той ще върне стойност **null**, но ние може предварително да проверим с **HasChanges(…)** дали има промени и дали има смисъл да викаме **GetChanges(…)**.

```
DataSet dsChanges = dsUsers.GetChanges();
```

4. Прилагаме направените промени като използваме метода **Update(…)** на **DataAdapter**. При внасяне на промените в източника могат да възникнат конфликти – например, ако след като сме заредили данните в **DataSet** обекта, някой изтрие запис в източника, а ние променим същия запис в извлечените данни, при обновяване ще настъпи конфликт.

   За разрешаване на конфликтите използваме обработчика на събитието **RowUpdated**. В него можем да разрешим директно конфликта или да си запазим грешките, които са възникнали. Ето пример:

```
userDataAdapter.RowUpdated +=
   new SqlRowUpdatedEventHandler(OnRowUpdated);
```

```
private void OnRowUpdated(object sender,
   SqlRowUpdatedEventArgs e)
{
   // Handle the conflict …
   e.Status = UpdateStatus.Continue;
}

userDataAdapter.Update(dsChanges);
```

5. Разрешаваме конфликтите. Това може да стане по различен начин, например чрез изтриване на конфликтните редове, чрез обновяване на конфликтната стойност, чрез задаване на неутрална стойност или чрез намеса на потребителя.

6. Зареждаме отново **DataSet** обекта с данни от базата, за да работим с актуални данни. Това се налага в случаите, когато други потребители са добавяли записи в базата след като сме заредили данните за последен път – тогава те няма да присъстват в нашия **DataSet**.

```
userDataAdapter.Fill(dsUsers);
```

Реализация на несвързан модел с DataSet и DataAdapter – пример

В настоящия пример ще реализираме стандартния сценарий за използване на несвързан модел. Ще видим как се използват **DataSet** и **DataAdapter** и как се разрешават конфликти по време на обновяване на базата данни. Ето как изглежда пълният сорс код на примера:

```
using System;
using System.Data;
using System.Data.SqlClient;

class DataAdapterTest
{
   static void Main()
   {
      string strCon = "Data Source=(local);" +
         "Integrated Security=SSPI;Database=Northwind";
      SqlConnection cnNorthwind = new SqlConnection(strCon);

      // Create and populate with data the table for the example
      CreateTableAlabala(cnNorthwind);
      PopulateTableAlabala(cnNorthwind);

      string strSelect = "SELECT Id, Name FROM Alabala";

      // Create an adapter and generate its commands
      // with a CommandBuilder
```

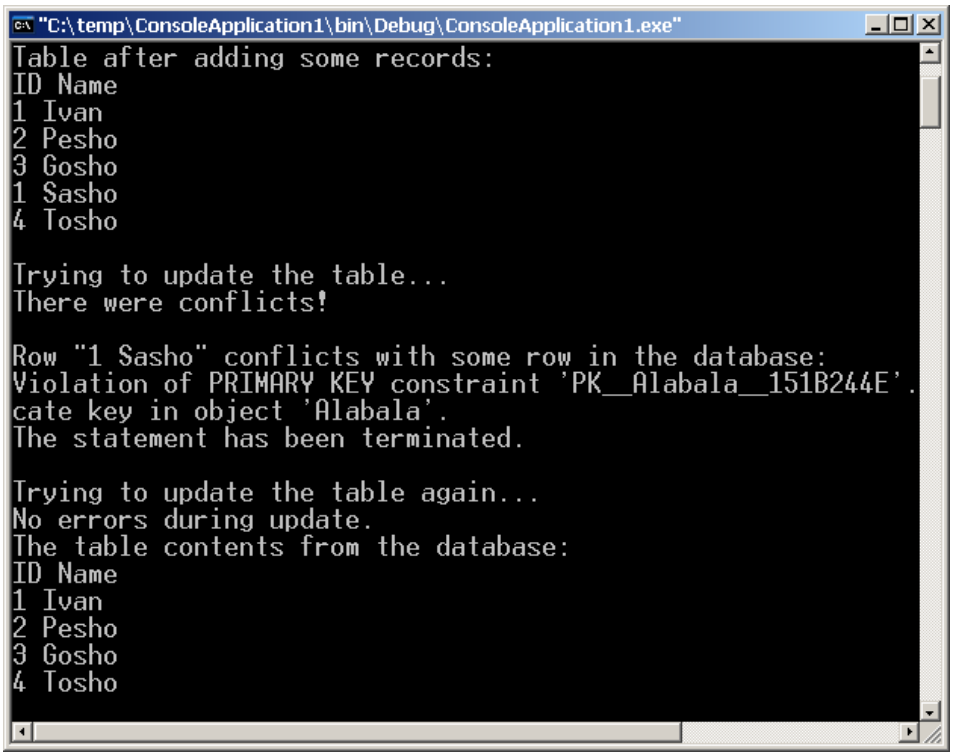
```
    SqlDataAdapter daAlabala = new SqlDataAdapter(
      strSelect, cnNorthwind);
    SqlCommandBuilder cbAlabala = new SqlCommandBuilder();
    cbAlabala.DataAdapter = daAlabala;

    // Add an event handler for the RowUpdated event
    daAlabala.RowUpdated += new SqlRowUpdatedEventHandler(
      OnRowUpdated);

    // Create and fill a DataSet
    DataSet dsNorthwind = new DataSet();
    daAlabala.Fill(dsNorthwind, "Alabala");

    Console.WriteLine("Initial table contents from the DB:");
    PrintTableAlabala(dsNorthwind);

    // Add some data to the DataSet
    MakeChanges(dsNorthwind);

    Console.WriteLine("Table after adding some records:");
    PrintTableAlabala(dsNorthwind);

    Console.WriteLine("Trying to update the table...");
    daAlabala.Update(dsNorthwind, "Alabala");

    // Check if there were conflicts and if so, resolve them
    // and call Update(...) again. This time there should be
    // no conflicts, because the conflicting rows are deleted
    // from the DataSet
    if (dsNorthwind.HasErrors)
    {
      Console.WriteLine("There were conflicts!");
      ResolveErrorsInTableAlabala(dsNorthwind);

      Console.WriteLine("Trying to update the table again...");
      daAlabala.Update(dsNorthwind, "Alabala");
      if (dsNorthwind.HasErrors)
        Console.WriteLine("Errors during update!");
      else
        Console.WriteLine("No errors during update.");
    }

    // Fill the DataSet with up-to-date data
    daAlabala.Fill(dsNorthwind);
    Console.WriteLine("The table contents from the database:");
    PrintTableAlabala(dsNorthwind);

    // Drop the sample table
    DropTableAlabala(cnNorthwind);
  }
```

```
// Add some rows to the DataSet
static void MakeChanges(DataSet aDataSet)
{
   // This row will cause a conflict - it has value
   // 1 for the Id field, but such value is already
   // present in the table, and Id is a primary key
   DataRow row = aDataSet.Tables["Alabala"].NewRow();
   row["Id"] = 1;
   row["Name"] = "Sasho";
   aDataSet.Tables["Alabala"].Rows.Add(row);

   // Add some non-conflict rows
   row = aDataSet.Tables["Alabala"].NewRow();
   row["Id"] = 4;
   row["Name"] = "Tosho";
   aDataSet.Tables["Alabala"].Rows.Add(row);
}

// RowUpdate event handler
static void OnRowUpdated(object aSender,
   SqlRowUpdatedEventArgs aEventArgs)
{
   // Check the Status property of the SqlRowEventArgs
   // argument for indication for errors
   if (aEventArgs.Status == UpdateStatus.ErrorsOccurred)
   {
      // Conflict found. Set error for the updated row
      string errorMessage = string.Format("\nRow \"{0} {1}\"" +
         " conflicts with some row in the database:\n{2}\n",
         aEventArgs.Row["Id"], aEventArgs.Row["Name"],
         aEventArgs.Errors.Message);
      aEventArgs.Row.SetColumnError("Id", errorMessage);

      // Set the Status property to Continue so that the
      // Update(...) method continues with the other rows
      aEventArgs.Status = UpdateStatus.Continue;
   }
}

// Resolves the conflicts for the rows that have errors
static void ResolveErrorsInTableAlabala(DataSet aDataSet)
{
   DataTable alabalaTable = aDataSet.Tables["Alabala"];
   DataRow[] conflictRows = alabalaTable.GetErrors();
   foreach (DataRow row in conflictRows)
   {
      DataColumn[] conflictColumns = row.GetColumnsInError();
      foreach (DataColumn column in conflictColumns)
      {
```

```
            string errorMessage = row.GetColumnError(column);
            Console.WriteLine(errorMessage);
        }

        // Resolve the conflict by deleting the conflicting row
        row.Delete();
        row.ClearErrors();
    }
}

// Prints the data from the DataSet on the console
static void PrintTableAlabala(DataSet aDataSet)
{
    Console.WriteLine("ID Name");
    foreach(DataRow row in aDataSet.Tables["Alabala"].Rows)
    {
        Console.WriteLine("{0} {1}", row["Id"], row["Name"]);
    }
    Console.WriteLine();
}

// Creates the table, used in the example
static void CreateTableAlabala(SqlConnection aConnection)
{
    try
    {
        aConnection.Open();
        string strCreate = "CREATE TABLE Alabala" +
            "(Id int primary key, Name nvarchar(15))";
        SqlCommand cmdCreate = new SqlCommand(strCreate,
            aConnection);
        cmdCreate.ExecuteNonQuery();
    }
    catch(SqlException sqlEx)
    {
        Console.WriteLine(sqlEx.Message);
    }
    finally
    {
        aConnection.Close();
    }
}

// Populates the created table with some sample data
static void PopulateTableAlabala(SqlConnection aConnection)
{
    try
    {
        aConnection.Open();
        SqlCommand cmdInsert = new SqlCommand();
```

```
        cmdInsert.Connection = aConnection;

        cmdInsert.CommandText = "INSERT INTO Alabala" +
          " VALUES(1, 'Ivan')";
        cmdInsert.ExecuteNonQuery();

        cmdInsert.CommandText = "INSERT INTO Alabala" +
          " VALUES(2, 'Pesho')";
        cmdInsert.ExecuteNonQuery();

        cmdInsert.CommandText = "INSERT INTO Alabala" +
          " VALUES(3, 'Gosho')";
        cmdInsert.ExecuteNonQuery();
      }
      catch(SqlException sqlEx)
      {
        Console.WriteLine(sqlEx.Message);
      }
      finally
      {
        aConnection.Close();
      }
    }

    // Drop the table at the end of the example
    static void DropTableAlabala(SqlConnection aConnection)
    {
      try
      {
        aConnection.Open();
        string strDropTable = "DROP TABLE Alabala";
        SqlCommand cmdDropTable =
          new SqlCommand(strDropTable, aConnection);
        cmdDropTable.ExecuteNonQuery();
      }
      catch(SqlException sqlEx)
      {
        Console.WriteLine(sqlEx.Message);
      }
      finally
      {
        aConnection.Close();
      }
    }
}
```

Ето как изглежда резултатът от изпълнението на примера:



Как работи примерът?

Първоначално създаваме тестова таблица **Alabala** и я запълваме с тестови данни. След това зареждаме таблицата **Alabala** в **DataSet** обект с помощта на **SqlDataAdapter**.

Локално, в таблицата от **DataSet** обекта добавяме няколко реда, някои от които ще предизвикват конфликт при обновяване.

Следва нанасяне на промените по данните в базата данни по стандартния начин – чрез **SqlDataAdapter**, за който са генерирани команди за обновяване на данните чрез **CommandBuilder**.

В обработчика на събитието **RowUpdated** за всеки ред, който предизвиква конфликт, използваме метода **SetColumnError(…)** на класа **DataRow**, за да зададем съобщение за грешка в него. Приемаме, че конфликтната колона е колоната **Id** (първичният ключ в таблицата). След отбелязване на грешката указваме, че искаме обновяването да продължи въпреки настъпилия конфликт.

Грешките се обработват в метода **ResolveErrorsInTableAlabala(…)**. В него извличаме от таблицата проблемните редове (за целта използваме метода **GetErrors()** на **DataTable**) и ги обхождаме в цикъл. За всеки ред извличаме колоните, съдържащи грешка (с метода **GetColumnsInError()** на **DataRow**) и за всяка колона отпечатваме грешката на конзолата. След

това разрешаваме конфликта по най-простия начаин – като изтриваме проблемния ред.

При следващото обновяване на базата данни вече няма конфликти.

Класът XmlDataDocument

Класът **XmlDataDocument** представлява своеобразен "мост" между релационното представяне на данни (под формата на **DataSet** обект) и йерархичното им представяне чрез класовете за работа с XML данни. Този клас се използва за синхронизация на **DataSet** и XML документ и позволява да работим с едновременно с двете представяния на едни и същи данни.

XmlDataDocument наследява **XmlDocument**, като съответно получава всичките му възможности и добавя към тях синхронизация на двете представяния на данните. Той представя данните като DOM дърво, но същевременно поддържа и релационен изглед. Синхронизацията между двата изгледа се осъществява автоматично. Ако променим данните през релационния изглед, осигурен от **DataSet** обект, DOM дървото на XML документа ще се промени също. Обратно, при промяна на DOM дървото, промяната се отразява и в **DataSet** обекта.

Не е задължително **DataSet** обектът да представя целия XML документ. Възможно е схемата на **DataSet** обекта да включва таблици и колони само за някои елементи на XML документа, като по този начин можем да избираме какво точно от DOM дървото да представим релационно.

Предимства на XmlDataDocument

XmlDataDocument ни позволява да използваме за едни и същи данни всички средства за обработка на XML (като обхождане на DOM дървото, използване на XPath и XSL трансформации) от една страна, и възможностите, които дава **DataSet** класа (например свързване с контроли в уеб и Windows форми). Това предоставя значителна гъвкавост на работата с данни в приложението.

Друго предимство е възможността да се запази напълно точността на XML документа. Ако заредим съдържанието на XML документ в **DataSet** и след това го запишем отново във файл, форматирането няма да се запази, и освен това ще загубим всички елементи, за които няма съответствие в схемата на **DataSet** обекта. При използване на **XmlDataDocument** този проблем не съществува. Той поддържа форматирането и йерархичната структура на оригиналния XML документ, а **DataSet** обектът представя само тези данни и информация за схемата, които искаме.

Начини за синхронизация

Съществуват различни начини да се извърши синхронизацията между **DataSet** и **XmlDataDocument**.

Единият вариант е първо да се създаде **DataSet**, да му се зададе схема и да се запълни с данни, а след това да се синхронизира с нов **XmlDataDocument**. Така се осигурява йерархичен изглед на съществуващи релационни данни.

Възможен е и друго вариант – да се създаде **DataSet** със схема, да се синхронизира с **XmlDataDocument**, след което в последния да се заредят данните от XML документ. Имената на таблиците и колоните в **DataSet** обекта трябва да съответстват на имената на елементите в XML документа, които искаме да бъдат синхронизирани с **DataSet** обекта. Така се предоставя релационен изглед на съществуващи йерархични данни.

Третият вариант е да се създаде **XmlDataDocument** и съдържанието му да се зареди от съществуващ XML документ, след което да се получи релационен изглед чрез свойството **DataSet** на **XmlDataDocument**. В този случай трябва да се зададе схемата на **DataSet** обекта преди да се използва за достъп до данни. Както и в предния случай, трябва да има съответствие на имената на таблиците и колоните и XML елементите.

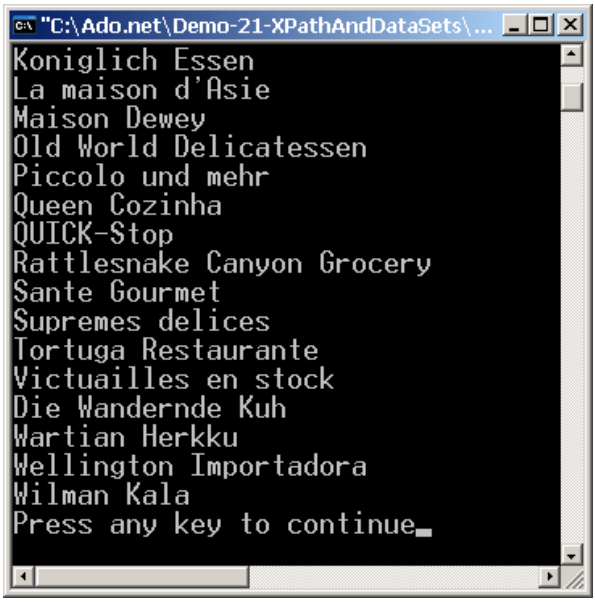
XmlDataDocment – пример

Следващият пример показва как можем да използваме **XmlDataDocument** за синхронизация на XML документ с **DataSet**:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Xml;

class XmlDataDocumentDemo
{
  static void Main()
  {
    DataSet myDataSet = new DataSet("CustomerOrders");

    SqlConnection nWindConn = new SqlConnection(
      "Data Source=localhost;Initial Catalog=northwind;" +
      "Integrated Security=SSPI;");
    using (nWindConn)
    {
      nWindConn.Open();

      SqlDataAdapter custDA = new SqlDataAdapter(
        "SELECT * FROM Customers", nWindConn);
      custDA.Fill(myDataSet, "Customers");

      SqlDataAdapter ordersDA = new SqlDataAdapter(
        "SELECT * FROM Orders", nWindConn);
      ordersDA.Fill(myDataSet, "Orders");
```

```
        SqlDataAdapter detailsDA = new SqlDataAdapter(
          "SELECT * FROM [Order Details]", nWindConn);
        detailsDA.Fill(myDataSet, "OrderDetails");
      }

      myDataSet.Relations.Add("CustOrders",
        myDataSet.Tables["Customers"].Columns["CustomerID"],
        myDataSet.Tables["Orders"].Columns["CustomerID"]).
        Nested = true;

      myDataSet.Relations.Add("OrderDetail",
        myDataSet.Tables["Orders"].Columns["OrderID"],
        myDataSet.Tables["OrderDetails"].Columns["OrderID"],
        false).Nested = true;

      XmlDataDocument xmlDoc = new XmlDataDocument(myDataSet);

      // Get all elements with ProductID=43
      XmlNodeList nodeList = xmlDoc.DocumentElement.SelectNodes(
        "descendant::Customers[*/OrderDetails/ProductID=43]");

      foreach (XmlNode myNode in nodeList)
      {
        DataRow customer = xmlDoc.GetRowFromElement(
          (XmlElement)myNode);
        Console.WriteLine(customer["CompanyName"]);
      }
    }
}
```

Следва изглед на част от изхода от примера:



В примера използваме първия начин за синхронизация. Първо създаваме **DataSet** обект **myDataSet** и го запълваме с данни от таблиците **Customers**, **Orders** и **Order Details** от базата **Northwind**. След това синхронизираме **myDataSet** с **XmlDataDocument** обекта **xmlDoc**. С използване на метода **SelectNode** на класа **XmlNode** се извлича всички клиенти, които са поръчвали продукт с **ProductID** 43. Условието се задава с XPath израз. Данните за всеки от тези клиенти се извличат в **DataRow** обект, след което се извежда името на компанията, за която работи съответният клиент.

Сигурността при работа с бази от данни

Сигурността при базите от данни има различни аспекти. В настоящата точка ще се спрем само на най-важните от тях, тъй като непознаването им може да доведе до сериозни проблеми в разработваните приложения.

Сигурност при динамични SQL заявки

Когато използваме SQL заявки, които трябва да включват стойност, въведена от потребителя, **в никакъв случай не трябва да сглобяваме заявката чрез долепване (конкатенация) на низове**. Така има опасност да станем жертва на атака от тип SQL injection, каквато видяхме в секцията за параметричните заявки. Решението на този проблем в ADO.NET е да се използват паратерични команди за изпълняваните SQL заявки.

Connection pooling и сигурност

Използването на connection pooling не позволява сигурността да се интегрира в базата. Това е така, защото за да функционира механизмът на connection pooling, се изисква низът за връзка да е идентичен за всички връзки от пула (т. е. един и същ за всички потребители).

За да бъде решен този проблем, сигурността трябва да се управлява от приложението, а не от базата данни. В този случай всички потребители се автентикират към базата данни с интегрирана автентикация (при която не се задава парола), а приложението им дава различни права според своя вътрешна логика.

Съхраняване на connection string

Съхранението на низа за връзка към базата е важно за сигурността. Той често съдържа информация за автентикация, с която някой може да злоупотреби. Низът за връзка трябва да може да се ползва от всички потребители на системата, но не трябва да е достъпен за тях, т.е. те не трябва да могат да го извличат в явен вид.

Бихме могли да съхраним низа за връзка в конфигурационния файл на приложението (**Web.config** при уеб приложения или **App.config** при настолни), но той е прост текстов файл и съхранението в него не е сигурно.

Ако изберем този вариант, или трябва да кодираме низа по надежден начин, или да изнесем паролите за достъп на друго място и да ги добавяме към низа по време на работа.

При връзка със SQL Server можем да използваме Windows автентикация и тогава в низа за връзка няма да се съхраняват потребителско име и парола.

Като допълнителна мярка за сигурност по подразбиране свойството **`Persist Security Info`** на низа за връзка има стойност **`false`**. В този случай информацията за парола и потребителско име не се запазва в свойството **`ConnectionString`** на обекта за връзка – тази информация се премахва веднага след отваряне на връзката.

Защитна стена

Важно е, когато инсталираме сървъра за базата данни при клиента, да го защитим от нежелан външен достъп (от Интернет и от локалната мрежа). Това може да стане чрез най-обикновена защитна стена (firewall), която филтрира портовете, на които сървърът очаква своите клиенти.

Препоръчваната практика е да бъде забранен достъпът до сървъра с базата данни за всички машини, включително и за машините от локалната мрежа, и да бъде разрешен достъп само за тези машини, които го използват директно.

Тази практика намалява значително рисковете сървърът да бъде атакуван и така повишава сигурността на информацията, която той съхранява.

Криптиране на комуникацията

Комуникацията между клиента и сървъра на базата данни обикновено не е криптирана и трафикът може да бъде подслушан. За повишаване на сигурността при обмяната на данни между клиента и сървъра може да се използва протоколът IPSec, който криптира целия мрежов трафик или да се използва VPN връзка, която също дава възможност за криптиране.

За сигурно предаване на данните, между отделни компоненти на дадено приложение може да се използва протоколът SSL. Друга възможност е да се криптират чрез публичен или споделен ключ данните при предаване на **`DataSet`** обекти по мрежата.

Упражнения

1. Обяснете какво представляват свързаният и несвързаният модел за достъп до данни. Опишете в кои случаи се използва единият, и в кои – другият. Дайте примери. Опишете предимствата и недостатъците на двата модела.

2. Опишете еднослойните, двуслойните, трислойните и многослойните приложения – какво представляват съответните модели, техните предимства и недостатъци, случаи на използване. Дайте примери.
3. Обяснете накратко какво е ADO.NET и кои са неговите основни пространства от имена. За какво служат те?
4. Какво представляват доставчиците на данни и кои са стандартните доставчици на данни в ADO.NET?
5. Опишете класовете от **`SqlClient`** доставчика на данни.
6. Опишете начините за автентикация пред MS SQL Server. Дайте пример за символен низ за връзка с SQL Server (connection string)
7. Опишете механизма на connection pooling. Защо е необходимо да се използва?
8. Обяснете какво представлява свързаният модел за достъп до данни.
9. Обяснете кои са основните класове и интерфейси за работа със свързан модел в ADO.NET.
10. Напишете програма, която извлича от базата данни **`Northwind`** в SQL Server всички категории продукти и имената на продуктите от всяка категория. Използвайте таблиците **`Categories`** и **`Products`**.
11. Напишете процедура, която добавя нов продукт в таблицата с продуктите в базата данни **`Northwind`** в SQL Server. Използвайте параметрична SQL заявка.
12. Разглеждаме проста система за обслужване на банкомат. Създайте нова база данни **`ATM`** в MS SQL Server за съхранение на сметките на картодържателите и наличностите по тях. Добавете нова таблица **`CardAccounts`**. В таблицата дефинирайте следните полета: **`Id`**, **`CardNumber`**, **`CardPIN`**, **`CardCash`** (като на картинката по-долу):

	Column Name	Data Type	Length	Allow Nulls
▶	Id	bigint	8	
	CardNumber	char	10	
	CardPIN	char	4	
	CardCash	money	8	

Добавете няколко примерни записа в таблицата (ще ви трябват за тестване). Използвайки транзакции напишете процедура, която тегли дадена сума (например 200 лв.) от дадена картова сметка. Операцията по тегленето на пари се изпълнява успешно когато е налице успешното изпълнение на следната поредица от съставни операции:

- Проверява се със заявка дали подаденият пин-код (**`CardPIN`**) съответства на номера на картата (**`CardNumber`**).
- Проверява се наличността (**`CardCash`**) по картовата сметка дали е повече от заявената сума (повече от 200 лв.).

- Банкоматът изплаща заявената сума (200 лв.) и записва в таблицата **CardAccounts** новата наличност (**CardCash = CardCash – 200**).

Ако някоя от съставните операции се провали, банкоматът не изплаща нищо и отказва транзакцията. Трябва да реализирате процеса със средствата на ADO.NET без да използвате съхранени процедури в базата данни.

13. Разширете проекта от предната задача и добавете нова таблица **AccountTransactions** с полета (**Id**, **CardId**, **TransactionDate**, **Ammount**), съдържаща информация за всички тегления на пари по всички сметки. Променете процедурата за теглене на пари, така че да запазва информация в новата таблица за всяко успешно извършено теглене. Съобразете се с необходимостта от използване на транзакция за цялата операция по тегленето. Добавете процедура, която по дадена карта и ПИН код показва списък на всички тегления, сортирани по дата. Съобразете се с препоръките за правилна работа с дати.

14. Създайте база данни в SQL Server за съхранението на библиотека с филми и видеоклипове. Напишете програма, която записва и чете филми от базата данни. Имайте предвид, че филмите са обемни файлове (най-често около 700 MB) и трябва да се вкарват и извличат от базата данни на части.

15. Опишете класовете, които се използват за реализация на несвързания модел в ADO.NET. За какво служи всеки от тях?

16. Каква е разликата между силно типизиран и нетипизиран **DataSet**? Как се създават двата вида **DataSet**?

17. Кои класове се използват при работа с **DataSet**? За какво служи класът **DataTable**? Как се добавят редове и колони в **DataTable**? За какво служи класът **DataRelation**? Какви са основните приложения на **DataView**? Как се добавят ограничения в **DataSet**? За какво служат потребителските изрази в колоните на таблиците?

18. Опишете средствата за зареждане на **DataSet** от XML документ и записване на съдържанието на **DataSet** в XML документ.

19. Създайте нова база от данни **University** през Enterprise Manager на SQL Server. Създайте в нея таблица **Students** със следната схема: **Students(Id int identity primary key**, **FirstName nvarchar(15) not null**, **LastName nvarchar(20) not null**, **Age int**, **TimeRecordAdded datetime default GETDATE())**. Запълнете таблицата с малко примерни данни. След това напишете програма, която извлича данните от таблицата в **DataSet** обект и променя заредените данни, като добавя, изтрива и модифицира редове. След това данните трябва да се обновяват в базата. Използвайте **DataAdapter** с ръчно написани команди за обновяване на базата (без да използвате **CommandBuilder**). Обработете събитието **RowUpdate**, така че да се справите с евентуални

конфликти. Опитайте да създадете конфликти ръчно и да ги разрешите с програмна логика. Запишете съдържанието на **`DataSet`** обекта във файл **`students.xml`**.

20. Опишете предназначението на класа **`XmlDataDocument`**.
21. Опишете някои основни съображения относно сигурността при работа с базите от данни.

Използвана литература

1. Бранимир Гюров, Светлин Наков, Стефан Захариев, Лазар Кирчев, Достъп до данни с ADO.NET – http://www.nakov.com/dotnet/lectures/Lecture-13-ADO.NET-v1.01.ppt
2. MSDN Library – http://msdn.microsoft.com
3. Accessing Data with ADO.NET (.NET Framework Developer's Guide) - http://msdn.microsoft.com/library/en-us/cpguide/html/cpconAccessingDataWithADONET.asp
4. David Sceppa, Microsoft ADO.NET, Microsoft Press, 2002, ISBN 0-7356-1423-7
5. Francesco Balena, Programming Microsoft Visual Basic .NET (Core Reference), Mirosoft Press, 2002, 0-7356-1375-3 – Chapter 21: ADO.NET in Disconnected Mode – http://www.microsoft.com/mspress/books/sampchap/5199.asp

Глава 15. Графичен потребителски интерфейс с Windows Forms

Очаквайте във втори том.

Глава 16. Изграждане на уеб приложения с ASP.NET

Очаквайте във втори том.

Глава 17. Многонишково програмиране и синхронизация

Очаквайте във втори том.

Глава 18. Мрежово и Интернет програмиране

Очаквайте във втори том.

Глава 19. Отражение на типовете (Reflection)

Очаквайте във втори том.

Глава 20. Сериализация на данни

Очаквайте във втори том.

Глава 21. Уеб услуги с ASP.NET

Очаквайте във втори том.

Глава 22. Отдалечено извикване на методи (Remoting)

Очаквайте във втори том.

Глава 23. Взаимодействие с неуправляван код

Очаквайте във втори том.

Глава 24. Управление на паметта и ресурсите

Очаквайте във втори том.

Глава 25. Асемблита и разпространение

Очаквайте във втори том.

Глава 26. Сигурност в .NET Framework

Очаквайте във втори том.

Глава 27. Mono - свободна имплементация на .NET

Очаквайте във втори том.

Глава 28. Помощни инструменти за .NET разработчици

Очаквайте във втори том.

Глава 29. Практически проект

Очаквайте във втори том.

Заключение

Авторският колектив, изготвил настоящата книга, силно се надява, че тя ви е дала много нови и 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.

Ако имате въпроси или коментари, отправяйте ги в нашия форум:

http://www.devbg.org/forum/index.php?showforum=30

Главният автор на книгата и ръководител на проекта, Светлин Наков, отправя покана към всички, които желаят да изпробват описаните в тази книга технологии и да се научат да ги прилагат в практиката, да се запишат за 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.

Очаквайте скоро и втория том на книгата!

Светлин Наков,
септември, 2005

„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 954-775-505-6

Цена: 8,50 лв.

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
Светлин Наков
Стефан Добрев
Стефан Захариев
Стефан Кирязов
Стоян Дамов
Тодор Колев
Христо Дешев
Христо Радков
Цветелин Андреев
Явор Ташев