

Димитър Минчев

БУРГАСКИ СВОБОДЕН УНИВЕРСИТЕТ

ЖЪЛТА КНИГА ПО

С#

Издателство „Божич”

Настоящата "Жълта книга по C#" е ръководство по програмиране на езика C#, българско издание в превод на книгата "Yellow Book" на Rob Miles.

© Димитър Минчев

Жълта книга по C#

Някои права запазени. Разрешава се разпространението на книгата, ако това е с нетърговска цел. За издаване на книгата на хартиен носител или продажба на електронни копия на книгата трябва да имате договор с автора.

Издателство „Божич“ Бургас, 2013 г.

ISBN 978-954-9925-84-5

СЪДЪРЖАНИЕ

ОТЗИВ НА СВЕТИН НАКОВ	12
ПРЕДГОВОР	13
ГЛАВА 1. ВЪВЕДЕНИЕ И ТЕРМИНОЛОГИЯ	15
Какво е програмист?	15
Решаване на грешния проблем	15
Прост проблем.....	16
Получаване на заплащане.....	17
Участие на клиента.....	17
C#	18
Опасният C	18
Безопасният C#	18
C# и обектите	19
От какво се състои една програма на C#?	20
Съхраняване на данните.....	20
Описване на решение	21
Идентификатори и ключови думи	21
Обекти	22
ГЛАВА 2. ОСНОВИ НА ЕЗИКА.....	23
Първата ни програма на C#	23
using System;	23
class Windows.....	24
static.....	24
void.....	24
Main	25

()	25
{	25
double	25
width, height, woodLength, glassArea	26
;.....	26
string widthString, heightString;	26
widthString=	27
Console	27
ReadLine	27
()	27
;.....	28
width =.....	28
double.	28
Parse	28
(widthString);.....	29
heightString = Console.ReadLine(); height = double.Parse(heightString);	29
woodLength = 2*(width+height);.....	29
glassArea = width*height;	30
Console.WriteLine	30
(.....	30
"Необходимо количество дърво"	30
+.....	30
woodLength.....	31
+ " в метри"	31

)	32
;	32
}	32
}	32
Пунктуация	32
Променливи и данни	33
Съхранение на числа	33
Съхраняване на стойности на цели числа	34
Съхранение на стойности на реални числа	34
Съхранение на текст	36
Escape последователност	36
Кодова стойност на символите	37
Низови променливи	37
Запазване на състояние чрез bool	37
Идентификатори	38
Задаване на стойности на променливите	38
Изрази	39
Операнди	39
Оператори	39
Промяна на типа на данните, чрез разширяване и стесняване ...	40
Кастване	41
Кастване и буквални стойности	41
Типове данни в изразите	42
Писане на програма	43
Блокови коментари	43

Линейни коментари	44
Контролиране на програмния поток	44
Състояния и оператори за сравнение	44
==	44
!=	44
<	44
>	45
<=	45
>=	45
!	45
Комбиниране на логически оператори	45
&&	45
.....	45
Използване на блокове за комбиниране на изрази	46
Цикли	46
цикъл do ... while	46
цикъл while	47
цикъл for	47
Прекъсване на цикъл	48
Връщане обратно в началото на цикъл	49
Кратки оператори	49
i++	50
++i	50
Спретнато отпечатване	50
Настройване на точността на реално число	50

Уточняване на броя на отпечатани знаци	50
Наистина капризно форматиране.....	51
Отпечатване в колони	51
ГЛАВА 3. СЪЗДАВАНЕ НА ПРОГРАМИ	53
Необходимостта от методите	53
Параметри.....	54
Върнати стойности	54
Полезен метод.....	55
Подаване на параметър чрез стойност	56
Подаване на параметри чрез референция	56
Подаване на параметри като изходящи референции	57
Библиотеки от методи	58
Обхват на променливите.....	59
Обхват и блокове.....	59
Локални променливи за цикъла for	60
Член данни в класове.....	60
Масиви	61
Елементи на масива	62
Номериране на елементите в масива	62
Големи масиви	62
Създаване на двумерен масив.....	63
Повече от две дименсии.....	64
Изключения и грешки	64
Хващане на изключения	65
Обекта изключение.....	66

Добавяне на клаузата finally	67
Хвърляне на изключение	68
Конструкцията switch	68
Използване на файлове	70
Създаване на изходен поток	71
Писане в поток	71
Затваряне на поток	72
Потоци и пространства от имена	72
Четене от файл	72
Откриване на края на входен файл.....	73
Път до файла в C#	73
ГЛАВА 4. СЪЗДАВАНЕ НА РЕШЕНИЯ	75
Обхват на банковата система	75
Изброени типове	76
Създаване на тип enum.....	76
Проста структура.....	77
Създаване на структура.....	78
Използване на структура.....	79
Използване на тип структурапри извикване на метод.....	80
Обекти и структури	81
Създаване и използване на инстанция на клас	81
Множество референции към инстанция	83
Без референции към инстанция.....	83
Данни в обектите	84
Защита на данните в обектите	85

Публични методи	86
Завършен клас за сметки	88
Разработка базирана на тестове	88
Статични членове на клас	89
Използване на статични член данни в клас	90
Използване на статичен метод в клас	91
Използване на член данни в статични методи	91
Създаване на обекти	93
Конструктор по подразбиране	93
Предефиниран конструктор	94
Предефиниране на методи	94
Конструктора не може да се провали	94
От обект към компонент	95
Интерфейс и дизайн	95
Имплементиране на интерфейс	95
Референции към интерфейси	96
Използване на интерфейси	97
Имплементиране на множество интерфейси	97
Наследяване	98
Разширяване на родителски клас	99
Припокриване на методи	99
Виртуални методи	100
Защита на данните в йерархия от класове	101
Използване на базов метод	101
Конструктори и йерархии	102

Верижно свързване на конструктори	103
Абстрактни методи и класове	103
Референции към абстрактни класове	105
Проектиране на обекти и компоненти	105
Не изпадайте в паника!.....	105
Обектите и методът ToString	106
Класът Обект	106
Метод ToString	106
Обекти и тестване за равенство	107
Това е моят клас точка	108
Добавяне на собствени методи за равенство.....	108
Обекти и this.....	109
<i>this</i> като референция към текущата инстанция	109
Подаване на референция към себе си за други класове	110
Силата на низовете и символите.....	110
Сравняване на низове	110
Редактиране на низ	110
Дължина на низ	111
Главни и малки букви.....	111
Премахване и празни низове	111
Символни команди.....	112
Свойства	112
Свойства като член клас.....	112
Създаване на методите Get и Set.....	112
Използване на свойствата.....	113

Свойства и интерфейси	115
Използване на Делегати	115
ГЛАВА 5. ПРОГРАМИРАНЕ ЗА НАПРЕДНАЛИ	117
Списък	117
Нишки	118
Какво е нишка?	118
Защо имаме нишки?	119
Нишките и процесора	119
Добавяне на поддръжката на нишки в програмата	121
Укажете къде започва изпълнението на нишка	121
Създаване на нишка	121
Стартиране на нишка	121
Създаване на повече нишки	123
Нишки и синхронизация	125
Използване на взаимно изключване за управление на споделени данни	125
Контрол на нишки	126
Паузиране на нишки	127
Свързване на нишки	127
Контрол на нишки	127
Намиране на състоянието на нишка	128
ИЗТОЧНИЦИ	129

ОТЗИВ НА СВЕТЛИН НАКОВ

От години преподавам програмиране и съм се убедил, че първите стъпки при начинаещите са най-трудни. Полезно е, когато започнеш с книга, видео уроци или курс, където да се обяснят с простички думи и по достъпен начин основите на програмирането, базовите конструкции в езика и платформата, придружени с много примери, които лично да изпробваш.

"Жълта книга по C#" е ръководство по програмиране за начинаещи, което запознава с основите на програмирането с езика C# – типове данни, условни конструкции, цикли, методи, масиви, символни низове, изключения и файлове. Ръководството разглежда и основите на обектно-ориентираното програмиране (ООП) в C#, работа с класове, структури, интерфейси, наследяване и полиморфизъм, конструктори, виртуални методи, свойства, делегати и събития.

Учебното пособие е качествен превод на български език с адаптация на книгата "C# Programming – Yellow Book" на Rob Miles, направено на едно компетентно техническо ниво от Димитър Минчев, дългогодишен преподавател по програмиране и компютърни науки в Бургаски свободен университет. Препоръчвам книгата на всички, които искат да направят бързи начални стъпки в основите на програмирането.

Светлин Наков (вж. <http://www.nakov.com>) е софтуерен инженер с 20 години опит в програмирането и разработката на софтуер, преподавател, консултант, ръководител на проекти и предприемач. Той ръководи най-мощния образователен проект за обучение на софтуерни инженери в България – Софтуерна академия на Телерик (вж. <http://academy.telerik.com>), която дарява качествено обучение, професия и работа на хиляди млади хора в софтуерната индустрия.

ПРЕДГОВОР

Настоящата "Жълта книга по C#" е ръководство по програмиране на езика C#, тя е българско издание в превод на книгата "Yellow Book" на Rob Miles. Предназначена е за всички начинаещи в програмирането.

Първа глава включва въведение и терминология при работа с езика C#. Представено е от какво се състои една програма на C#. Направен е преглед на съхраняване на данни и описване на решение. Обърнато е внимание на идентификатори, ключови думи и запознаване с обектите.

Втора глава поставя основите на езика C#. Съдържа първата ни програма на C#. Обърнато е внимание на: пунктуация; променливи и данни; цели и реални числа, текстов низ и символи, булев тип; изрази, операнди и оператори; кастване; коментари; условни конструкции; конструкции за цикъл; печат и форматиране.

Трета глава продължава да развива програмните умения: запознавайки с необходимостта от методи, техните параметри и върнати стойности; разделяне на програмата на управляеми парчета манипулиране на голям обем данни използвайки масиви; обработка на грешки посредством хващане и хвърляне на изключение; използване на файлове, чрез изходни потоци за запис във файл и входни потоци за четене от файл.

Четвърта глава задълбочава познанията чрез: създаване и използване на структури; работа с класове и обекти, техните инстанции и референции; използване на компоненти, интерфейс и дизайн; наследяване; припокриване на методи; виртуални методи; конструктори и йерархии; абстрактни методи и класове; референции към абстрактни класове; проектиране на обекти и компоненти; свойства, интерфейси и делегати. Създава

се цяло банково приложение използвайки езика за програмиране C# и се разглеждат особеностите, които помагат да се разработва лесно.

Пета глава засяга програмиране за напреднали. Разгледана е структурата данни списък и нейната употреба. Запознаване с възможностите за многозадачна работа на процесора посредством употребата на нишки, включващо: добавяне на поддръжката на нишки; създаване и стартиране на нишка; създаване на повече нишки; синхронизация на нишките; контрол на нишки; намиране на състоянието на нишка.

ГЛАВА 1. ВЪВЕДЕНИЕ И ТЕРМИНОЛОГИЯ

Настоящата глава включва въведение и терминология при работа с езика C#. Представено е от какво се състои една програма на C#. Направен е преглед на съхраняване на данни и описване на решение. Обърнато е внимание на идентификатори, ключови думи и запознаване с обектите.

Какво е програмист?

Имате да решите дадена задача. На Ваше разположение е голяма торба с трикове, в случая език за програмиране. Разглеждате задачата известно време, измисляте начин да я решите и накрая свързвате частите на езика заедно. Изкуството на програмирането е познанието кои парчета се нуждаете да изкарате от Вашата торба с трикове, за да решите определена част от задачата.

Гледна точка на програмиста

Програмирането не е само математика, то е организиране и структуриране.

Решаване на грешния проблем

Най – лошото нещо, което можете да кажете на един клиент е: "Аз мога да направя това!". Вместо това Вие трябва да помислите: "Точно това ли изисква клиента?".

Спецификацията на функционалния дизайн казва точно какви са изискванията на клиента. Модерните техники за разработване на софтуер поставят клиента точно в центъра на разработването.

Вие, като разработчик не знаете много за бизнеса на клиента Ви, както клиента не знае ограниченията и възможностите на технологията. Като вземете това в предвид, добра идея е да направите серии от версии на задачата и да обсъдите всяка една

от тях с клиента, преди да продължите към изпълнението на следващата задача. Този процес се нарича **прототипиране**¹.

Прост проблем

Клиентът иска решаване на прост проблем, като въведе размерите на прозорец (*височина и широчина*) да получи разпечатка на материалите, необходими за направа на прозореца под формата на необходимото количество дърво и стъкло.

Гледна точка на програмиста

Метаданните са важни. Информацията относно информацията се нарича метаданни.

Пример:

стъкло **площ** = височина * широчина

дърво **дължина** = (височина + широчина) * 2

Клиентът трябва да разбере, че информацията, която подава на програмата трябва да бъде в определен обхват. Широчината на прозореца – в метри и със стойност между 0.5 и 3.5 метра, включително. Височината на прозореца, в метри и между 0.5 и 2.0 метра, включително.

Записвайки всичко това във вид, който както Вие така и клиентът Ви разбирате, създайте тест, който ще позволи да докажете, че програмата работи.

Примерен тест: „Ако въведе на горната програма стойността 2 метра височина и 3 метра широчина

¹ Прототипирането е процес на проектиране на софтуерен продукт, при който потребителят получава представа за функционалността на приложението. Термин на Английски език = **prototype, prototyping**, вж:

<http://en.wikipedia.org/wiki/Prototyping>

програмата трябва да изведе, че имам нужда от 6 квадратни метра стъкло и 10 линейни метра дърво.“

Тестването е много важна част от развитието на програмата.

Получаване на заплащане

Установете фазова система на плащане, така че да получавате пари, докато разработвате програмата. Когато процесът на проектиране и тестване е завършен, то няма двусмислие, което може да доведе до искане за промени от страна на клиента. Добрата новина за разработчика е, че ако все пак се изискат промени, те могат да бъдат разглеждани в контекста на допълнителна работа, за която може да се очаква заплащане.

Участие на клиента

Точното взаимодействие с потребителя (какво прави програмата, когато срещне грешка, как информацията бива представена и т.н.) е диалог, за който гарантирано клиента трябва да има силно изразено мнение. В идеалния случай цялата тази информация трябва да бъде описана подробно, както и да бъдат включени изгледи на екраните и детайли за това, кои клавиши ще бъдат натискани за всяко едно действие. Доста често се използва прототипиране, за да се добие представа как програмата трябва да изглежда и работи.

Гледна точка на програмиста

Добрите програмисти са добри комуникатори с клиентите. Да говориш с клиента и да намериш точно какво той или тя иска е изкуство. Едно от първите неща, които трябва да направите е да си изкарате от главата идеята: „Аз пиша тази програма за теб!“ и да я замените с: „Ние заедно създаваме решение на проблема!“.

C#

Ние с Вас ще изучаваме програмен език наречен C# (*произнася се „СИ ШАРП“*). C# е много гъвкав и силен програмен език с интересна история. Той е разработен от компанията *Microsoft Corporation*. C# прилича силно на програмните езици C++ и Java, като е взел назаем (*подобрил*) функции, които тези езици изпълняват. Произходът на Java и C++ може да бъде проследен назад във времето до език наречен C, който е много опасен и забавен език, разработен в периода 1969 – 1973.

Опасният C

Ако направя нещо глупаво C няма да ме спре, така че имам много по-голям шанс да забия програмата, която програмирам на съответния компютър, отколкото ако работя с по – безопасен език.

Безопасният C#

Програма на C# може да съдържа управлявани и не управлявани части от програмен код. **Управляваният код**² е зависим от компютърната система, която го стартира. Това осигурява трудно (*но не и невъзможно*) забиване на програмата на компютъра, който обработва управлявания код.

За да получим максимална възможна производителност и да позволим директен достъп до части от по – долните слоеве на компютърната система, можем да използваме **неуправляван**

²Управляваният код не се компилира до машинен код, а до междинен език, който се интерпретира и изпълнява от услугана компютъра. По този начин някой друг се грижи за управлението на паметта и сигурността, като предоставя безопасни условия за работа на програмата. Термин на Английски език = **Managed code**, вж. http://en.wikipedia.org/wiki/Managed_code

код³. Програма написана с неуправляван код работи по – бързо, но ако възникне грешка при изпълнението ѝ има шанс да отнесе и компютъра със себе си (*забиване или увисване на компютъра*).

C# и обектите

Езикът C# е **обектно-ориентиран**⁴. Обектно-ориентираният дизайн прави големи продукти много по – лесни за разработка, тестове и разширяване. Също така позволява създаването на програми, които имат голяма степен на надеждност и стабилност.

C# е компилаторен програмен език. Компютърът не може да разбере езика директно, за това програма, наречена компилатор преобразува текста на програмата написана на езика C# в инструкции от ниско ниво. Тези инструкции от ниско ниво задвижват хардуера, който стартира Вашата програма.

Компилаторът⁵ е много голяма програма, която знае да преценя дали дадена програма е правилна. Първо той проверява за грешки. Компилаторът ще изведе резултат, само ако не намери грешки в програмата. Той също така докладва с

³Неуправляваният код се компилира до машинен код и следователно се изпълнява от операционната система директно. Тогава програмиста е този, който трябва да се погрижи за управлението на паметта и сигурността на програмата. Термин на Английски език = **Unmanaged code**, вж. <http://msdn.microsoft.com/en-us/library/ms973872.aspx>

⁴Обектно-ориентираното програмиране е програмна парадигма, която представлява работа с обекти (инстанции на класове), притежаващи частни данни, достъпни посредством публични методи. Термин на Английски език = **Object-oriented programming**, вж. https://en.wikipedia.org/wiki/Object-oriented_programming

⁵Компилатор е компютърна програма, която преобразува изходния код написан на език за програмиране в изпълним код. Термин на Английски език = **Compiler, Compiling**, вж. <http://en.wikipedia.org/wiki/Compiler>

предупреждения, когато забележи, че сте направили нещо, което не е технически грешно, но може да направи някъде проблем.

Компанията Microsoft предоставят инструмент наречен *Visual Studio*, който е страхотно място за писане на програми. Той се състои от компилатор, интегриран редактор и дебъгер. Налична е безплатна версия наречена **Visual Studio Express Edition**⁶, която е идеалното място да започнете. Друг безплатен ресурс е **Microsoft .NET Framework**⁷.

От какво се състои една програма на C#?

Можем да си мислим за една програма като за готварска рецепта, но написана за изпълнение от компютър. Съставките, които искате Вашата програма да използва, ще бъдат стойности (*наричани още променливи*). Самата програма ще бъде последователност от действия или стъпки (*наричани още програмни конструкции*), които ще изискваме компютъра да следва. Но вместо да пишем програмата на лист хартия, както при рецептата, я записваме във файл на компютъра, често наричан сорс файл (*сурс файълът съдържа сорс код*).

Съхраняване на данните

Променлива⁸ наричаме наименувано местоположение, в което се съдържа стойност, докато програмата работи. C# също така ни позволява да създаваме структури, които могат да съдържат цялата нужна информация за нормална работа на програмата.

⁶вж. <http://www.microsoft.com/visualstudio/eng/products/visual-studio-express-products>

⁷вж. <http://www.microsoft.com/net/>

⁸Променлива е място в паметта на компютъра за съхранение на информация. Тя се характеризира с идентификатор и стойност. Термин на Английски език = **Variable**, вж. [http://en.wikipedia.org/wiki/Variable_\(computer_science\)](http://en.wikipedia.org/wiki/Variable_(computer_science))

Описване на решение

Една единствена, проста инструкция да се направи нещо в програма на С# се нарича **програмна конструкция**⁹. В С# можете да групирате програмни конструкции заедно, като така получавате **блок**¹⁰ в програмата, който може да извършва определена задача. Такива наименовани блокове се наричат **методи**¹¹. Един метод може да бъде много малък или много голям. Позволено е да върне стойност, която ни интересува или не чак толкова. Методът има каквото си поискате име. Вашата програма е възможно да съдържа толкова методи, колкото смятате за добре. Случва се един метод да препраща към други. Езикът С# също така има огромен брой библиотеки на свое разположение, които е позволено да използвате. Те Ви спасяват от „*преоткриване на колелото*“ (или *преоткриване на топлата вода*) всеки път, когато пишете нова програма.

Идентификатори и ключови думи

Езикът С# всъщност обработва Вашата програма, като търси метод със специално име наречен *Main*. Този метод бива извикан и стартиран, когато програмата започне да работи и спира своята работа, когато *Main* завърши. Имената, които Вие измисляте за да идентифицирате обекти или променливи, се

⁹Програмната конструкция е най-малкия самостоятелен елемент в езика за програмиране. Програмата се състои от последователност от програмни конструкции. Термин на Английски език = **Statement**, вж.

[http://en.wikipedia.org/wiki/Statement_\(computer_science\)](http://en.wikipedia.org/wiki/Statement_(computer_science))

¹⁰Блокът е групиран программен код, който се състои от една или повече програмни конструкции. Термин на Английски език = **Block, Lump**, вж.

[http://en.wikipedia.org/wiki/Block_\(programming\)](http://en.wikipedia.org/wiki/Block_(programming))

¹¹Методите са наименовани програмни фрагменти състоящи се от блокове. Термин на Английски език = **Methods**, вж.

[http://en.wikipedia.org/wiki/Method_\(computer_programming\)](http://en.wikipedia.org/wiki/Method_(computer_programming))

наричат **идентификатори**¹². Думите, които са част от самия език C#, се наричат **ключови думи**¹³.

Обекти

Някои неща в програмите, които пишем са **обекти**¹⁴, те са част от **работната рамка**¹⁵, която използваме или са наши собствени обекти.

¹²Идентификаторите в компютърните езици са имена на променливи, методи, класове и др. Термин на Английски език = **Identifiers**, вж. <http://en.wikipedia.org/wiki/Identifiers>

¹³Ключови думи са термините, които езика за програмиране използва. Термин на Английски език = **Keywords**, вж. <http://en.wikipedia.org/wiki/Keywords>

¹⁴Обектите са инстанции на класовете. Термин на Английски език = **Objects**, вж. [http://en.wikipedia.org/wiki/Object_\(computer_science\)](http://en.wikipedia.org/wiki/Object_(computer_science))

¹⁵Работната рамка е готова универсална платформа, съдържаща програмен код за многократно използване при разработването на компютърни програми. Термин на Английски език = **Framework**, вж. http://en.wikipedia.org/wiki/Software_framework

ГЛАВА 2. ОСНОВИ НА ЕЗИКА

Настоящата глава поставя основите на езика C#. Съдържа първата ни програма на C#. Обърнато е внимание на: пунктуация; променливи и данни; цели и реални числа, текстов низ и символи, булев тип; изрази, операнди и оператори; кастване; коментари; условен оператор; оператори за цикли; печат и форматиране.

Първата ни програма на C#

Първата програма, която ще разгледаме ще чете широчината и височината на прозорец и ще отпечата необходимото количество дърво и стъкло за направата на прозорец, който ще пасне на дупка в стената с тези размери.

```
using System;
class windows
{
    static void Main()
    {
        double width, height, woodLength, glassArea;
        string widthString, heightString;
        widthString = Console.ReadLine();
        width = double.Parse(widthString);
        heightString = Console.ReadLine();
        height = double.Parse(heightString);
        woodLength = 2*(width+height);
        glassArea = width*height;
        Console.WriteLine("Необходимо количество дърво "+woodLength+" в метри");
        Console.WriteLine("Необходима площ стъкло "+glassArea+" в квадратни метри");
    }
}
```

using System;

Това е инструкция към компилатора на C#, с която му казваме, че искаме да използваме неща от **пространството от имена**¹⁶,

¹⁶Пространствата от имена са контейнери за идентификатори, групирани по тяхната функционалност. Термин на Английски език = **Namespace**, вж.

<http://en.wikipedia.org/wiki/Namespace>

наречено *System*. Пространството от имена е място, където точно определени имена (*наричани още: термини*) имат значение. В C# пространството от имена наречено *System*, са описани много полезни неща. Едно от тях, което е на разположение на програмиста е обекта *Console*, който ни позволява да пишем информация, която ще се появява на екрана на потребителя.

class Windows

Класовете са базис на обектно-ориентираното програмиране. Една програма на C# е съставена от един или повече класове. Класът е контейнер, който съдържа данни и програмен код, за да извърши определена работа. Класът в случая съдържа само един метод. Трябва да измислите идентификатор за всеки клас, който създавате. Нарекох нашия клас *Windows*, понеже това наименование отразява какво програмата прави. Има установена практика, че името на файла, който съдържа определен клас трябва да съответства на името на самия клас, с други думи горната програма трябва да бъде записана във файл наречен *Windows.cs*.

static

Тази ключова дума гарантира, че методът, който следва винаги присъства, тоест думата *static* в този контекст означава: това е статичен метод и такъв трябва да бъде.

void

Void (празнота) е нищо. На програмен език ключовата дума *void* означава, че методът, който сме на път да опишем няма да върне нищо. Методът просто ще извърши своята работа и ще приключи. В някои случаи пишем методи, които връщат резултат, както ще видим по-късно.

Main

Избирайте имената на методите имайки в предвид какво ще правят те за Вас. С изключение на метода *Main*. Този метод (*трябва да има един и само един такъв метод*) е там, където програмата започва да работи. Когато програмата Ви бъде заредена и стартира, първият метод, на който се дава контрол е точно този метод, наречен *Main*. Ако пропуснете метод *Main*, системата буквално не знае от къде да започне своето изпълнение.

()

Това е чифт скоби, ограждащи нищо. Те казват, че методът *Main* няма параметри. Когато дефинирате метод, можете да кажете на C#, че той работи с един или повече параметри, например $\sin(x)$ може да работи със стойността на параметъра на ъгъла x , зададен като число с плаваща запетая.

{

Къдравите скоби идват по двойки, тоест за всяка отваряща къдрава скоба трябва да има съответна затваряща. Къдравите скоби позволяват на програмистите да смесват парчета от програмата заедно. Тези парчета на програма се наричат **блокове**. В този случай къдравите скоби ограждат програмния фрагмент на метода *Main*.

double

Нашата програма има нужда да запомня определени стойности по време на своята работа. В C# местата, където се съдържат стойностите, се наричат **променливи**. В началото на всеки блок, можете да кажете на C#, че искате да резервирате място в паметта, което да съдържа стойности за данни. Всяка клетка може да съдържа определен вид стойност. В основата си C#

може да борава с три типа данни, числа с плаваща запетая, цели числа и текст. Процесът на създаване на променлива се нарича **деклариране на променлива**.

`width, height, woodLength, glassArea`

Това е списък. В С# списъците от променливи се отделят със знак запетая. В случая това е списък с имена на променливи. Веднага след като компилатора види думата ***double***¹⁷ (*виж по – горе*), той очаква да намери името на поне една променлива, която да бъде създадена. Компилаторът обработва списъка, създавайки клетки, които да съдържат стойността от типа *double* и им дава подходящи имена. От тук нататък можем да използваме дадените имена и компилаторът ще знае, че искаме точно тази определена променлива.

;

Точката и запетая маркират края на списъка с имена на променливи и също така края на програмна конструкция за деклариране. Всички програмни конструкции в програма на С# са разделени със символа ;. Символът; е всъщност изключително важен. Той казва на компилатора, къде дадена програмна конструкция свършва. Ако компилаторът не намери точка и запетая, където очаква, то той ще изведе грешка.

`string widthString, heightString;`

Направихме няколко променливи, които могат да съдържат числа. Сега ще направим други, които да съдържат текстов низ. Ще направим това, защото когато четем числата, които

¹⁷Число с плаваща запетая и двойна точност. Фраза на Английски език = **Double-precision floating-point format**, вж. http://en.wikipedia.org/wiki/Single-precision_floating-point_format и <http://msdn.microsoft.com/en-us/library/678hzkk9.aspx>

потребителя на програмата въвежда, първоначално ние ги четем като низ от текст. По-късно ще променим текста в число.

`widthString=`

Това е конструкция за присвояване. В тази програмна конструкция на C# ще сменим стойността, която е записана в променливата. Нашата програма ще прочете ред текст въведен от потребителя чрез клавиатурата и ще постави резултата в променлива, която е наречена *widthString*.

Console.

В дясната страна на равенството имаме нещото, което ще бъде присвоено на *widthString*. В този случай резултатът ще бъде стринга върнат от метода *ReadLine*. Този метод е част от обект наречен *Console*, който обработва потребителския вход и изход. Точката (.) разделя обектния идентификатор от този на метода.

ReadLine

Извиква се методът *ReadLine*. Работещата програма бива препратена към метода, който изпълнява програмните конструкции в него, връща се обратно към програмата и продължава с нейното изпълнение. В случая се изчаква потребителя да въведе ред с текст и да натисне клавиша *Enter*. Всичко, което бъде написано от потребителя, ще бъде върнато като текст от метода *ReadLine*. Резултатът от извикването на метода е присвоено в променливата *widthString*. C# съдържа доста такива вградени методи, които правят различни неща за нашите програми.

()

След извикването на метод следват входните параметри на този

метод. **Параметър**¹⁸ е специален вид променлива, която се използва за подаване на входни данни към метод, за да може той да работи с тях. Винаги трябва да бъде предоставен лист от параметри, дори и той да е празен.

;

Вече сме виждали точката и запетая преди.

width =

Това е друга програмна конструкция за присвояване.

double.

Ще поискаме г-н **Double** (отговорен за съхраняването на числа с плаваща запетая и двойна точност) да свърши малко работа за нас. В този случай, задачата е „вземи низ-а, записан в *widthString* го превърни в число с плаваща запетая и двойна точност“. Г-н **Double** извършва поставената му задача, използвайки метод наречен **Parse**.

В същността си множеството C# библиотеки е съставено от серии от методи и едно от предизвикателствата пред Вас, с които трябва да се преборите, всъщност е къде са методите и как да ги използвате.

Parse

Работата, която метода *Parse* извършва е да преобразува текстовият низ, който му бъде подаден, като входен параметър в число с плаваща запетая и двойна точност. За тази цел, той трябва да обиколи низа от начало до край, да извлече всяка поредна цифра и след това да формира реалната цифрова стойност на съответното число, например „25“ означава

¹⁸Параметър с специален вид променлива, която се използва за подаване на входни данни към методите. Входните данни се наричат аргументи. Термин от Английски език = **parameter**, вж.

[http://en.wikipedia.org/wiki/Parameter \(computer programming\)](http://en.wikipedia.org/wiki/Parameter_(computer_programming))

двадесет и пет. Процесът на последователна обработка в този контекст се нарича **парсване**¹⁹.

Забележете, че това дава значителен потенциал за злодеяния, ако например потребителят не напише стойността на числото, а вместо това напише словом „Двадесет и пет“, методът Parse няма да може да върне число и ще се провали в задачата си за даване на резултат.

```
(widthString);
```

Видяхме, че извикване на метод трябва да бъде последвано от параметри за този метод. В случая с *Parse*, методът има нужда да му бъде даден низ, с който да работи. Стойността на информацията в *widthString* е подадена, като входен параметър на метода *Parse*, за да работи с нея и да изведе число.

```
heightString = Console.ReadLine(); height =  
double.Parse(heightString);
```

Тези две програмни конструкции просто повтарят процеса на четене на текст и преобразуването му в число с плаваща запетая и двойна точност.

```
woodLength = 2*(width+height);
```

Това е частта, която всъщност върши работата. Тя взема стойностите на височината и на широчината на прозореца и ги използва, за да изчисли необходимата дължина на дървото в линейни метри. Нормално C# би работил с изрази по начина, по който бихте очаквали, тоест първо ще бъдат изпълнени всички операции по умножение и деление, след което ще се изпълнят

¹⁹Парсването е процес на синтактичен анализ на низ от символи в съответствие с правилата на формалната граматика. Термин на Английски език = **parse**, **parsing**, вж. <http://en.wikipedia.org/wiki/Parsing>

операциите по събиране и изваждане. В горния израз някои операции са извършени приоритетно, за да се направи изчислението, както в математиката – са сложени кръгли скоби около операциите, които трябва да бъдат извършени първо.

Символите + и * в израза се наричат **оператори**, в смисъл, че те пораждат извършването на операция. Останалите елементи в израза се наричат **операнди**.

```
glassArea = width*height;
```

Този ред повтаря изчисленията, но за лицето в квадратни метри на необходимото количество стъкло за изработка на прозореца.

```
Console.WriteLine
```

Тази част извиква метод, точно както при метода *ReadLine*, с разликата, че този код взема, това което му бъде подадено като входни данни и го отпечатва на конзолата.

```
(
```

Тази скоба показва старта на параметрите, които ще бъдат подадени на метода при извикване. Вече сме виждали това при извикването на метода *Parse* също и на метода *ReadLine*.

```
"Необходимо количество дърво"
```

Това е низ от текст, който седи в програмата буквално както е написан. Низът от текст е заграден със знаци за двойни кавички, за да се каже на компилатора, че това е част от стойност в програмата, а не инструкция към самия компилатор.

```
+
```

Плюсът е оператор за добавяне. Видяхме как той е използван за събиране на две цели числа. В този случай обаче, той означава „*слепа два низа заедно*“. Това означава, че ще изпълни

конкатенация²⁰ на низове, вместо математическо събиране.

woodLength

Това е друг пример за работа в контекст. Много е важно да разберете точно какво се случва тук. Вземете под внимание следния програмен фрагмент:

```
Console.WriteLine(2.0+3.0);
```

Това ще изпълни цифрово изчисление (2.0 + 3.0) и ще изведе като резултат стойност с плаваща запетая и двойна точност. След което се създава низова версия, която бива отпечатана на екрана давайки следния резултат:

```
5
```

но следния програмен фрагмент:

```
Console.WriteLine("2.0"+3.0);
```

ще разгледа оператора + , като операция за конкатенация на два низа. Тогава това ще отпечата следния различен резултат:

```
2.03
```

Мислете следното, за всичките променливи в нашата програма е наличен етикет с метаданни, които компилатора използва, за да реши какво да прави с тях. Променливата *heightString* има етикет с информация, на който пише „това е низ, използвайки операция плюс конкатенирай“. Променливата *woodLength* има етикет, на който пише „това е променлива с плаваща запетая и двойна точност, използвайки операция плюс изпълни аритметика.“.

+ " в метри"

Друга конкатенация.

²⁰Конкатенацията представлява слепване или залепяне на един текстов низ към друг. Термин на Английски език = **concat, concatenate**, *вж.*

<http://en.wikipedia.org/wiki/Concatenate>

)

Тази скоба маркира края на конструирането на входния параметър, който метод *WriteLine* ще използва при извикването му.

;

Точката и запетая маркира края на тази програмна конструкция.

}

Програмата като цяло е завършена. Тази първа затваряща къдрава скоба маркира края на блока от код, който е тялото на метод *Main*. Блок от код винаги започва с { и завършва с }. Когато компилатора го види, той си казва „това е края на метода *Main*“.

}

Втората затваряща къдрава скоба маркира края на класа *Windows*. В C# всичко съществува в рамките на класовете. Класът е контейнер за толкова много неща, включително методи. Използваме втората затваряща къдрава скоба за да маркираме края на самия клас.

Пунктуация

Едно от нещата, които вероятно сте забелязали е, че се използва страшно много пунктуация. Бързо ще свикнете да претърсвате и забелязвате компилационни грешки. Ще забележите, че компилаторът не винаги улавя грешка там където няма. Друго, което трябва да запомните е, че оформлението на програмата не притеснява компилатора, следния програмен код е също толкова валиден:


```
using System; class windows { static void Main() { double width, height, woodLength, glassArea; string widthString, heightString; widthString = Console.ReadLine(); width = double.Parse(widthString); heightString = Console.ReadLine(); height = double.Parse(heightString); woodLength = 2*(width+height); glassArea = width*height; Console.WriteLine("Необходимо количество дърво "+woodLength+" в метри"); Console.WriteLine("Необходима площ стъкло " +glassArea+" в квадратни метри"); }}
```

Гледна точка на програмиста:

Оформлението на програмата е много важно. Програмните конструкции в скобите са умишлено сложени така, за да е ясно към какво са свързани (т.е. на какво принадлежат). Правим това така, защото иначе сме безпомощни да прочетем програмата и да разберем, какво тя прави. Намирам трудно да прочета програма, ако не е оформена правилно.

Променливи и данни

Променлива е именувано място в паметта, където може да съхранявате данни по време на работата на програмата. Може да си мислите за нея като кутия с определен размер и име написано на кутията. Можете да изберете име, отразяващо какво ще бъде съхранено в нея. Можете също така да изберете типа на променливата (*специфичния размер и форма на кутията*) от наличния избор на типове данни, които C# осигурява. Типът на променливата е част от мета данните относно променливата.

Съхранение на числа

Когато пишете със спецификации трябва да помислите относно прецизността, с която стойностите в променливите ще бъдат съхранявани. Твърде голяма точност може да забави компютъра – твърде малка, може да върне в резултат използването на грешни стойности.

Това означава, че когато искаме да съхраним информация, трябва да кажем на компютъра дали това е цяло или реално число. Също така трябва да вземем под внимание обхвата на възможните стойности, които съхраняваме, така че да изберем подходящия тип за тези данни. Редно е да кажете на C# каква променлива искате да създадете чрез декларирането ѝ. Декларацията също така идентифицира типа на информацията, която искаме да съхраним.

Съхраняване на стойности на цели числа

C# осигурява няколко типа за цели числа, в зависимост на обхвата на стойностите, които искате да съхраните в тях:

<i>sbyte</i>	8 бита	-128 до 127
<i>byte</i>	8 бита	0 до 255
<i>short</i>	16 бита	-32768 до 32767
<i>ushort</i>	16 бита	0 до 65535
<i>Int</i>	32 бита	-2147483648 до 2147483647
<i>uint</i>	32 бита	0 до 4294967295
<i>long</i>	64 бита	-9223372036854775808 до 9223372036854775807
<i>ulong</i>	64 бита	0 до 18446744073709551615
<i>char</i>	16 бита	0 до 65535

Съхранение на стойности на реални числа

Реалните числа имат десетична точка и дробна част. Стандартна **float** стойност се намира в интервала от 1.5E-45 до 3.4E+38 с точност само от 7 знака. Ако искате по – голяма точност (макар, че разбира се, програмите Ви ще използват по – голяма част от паметта на компютъра и ще работят по – бавно) може да използвате двойна кутия, вместо това (**double** е съкращение от *double precision*, двойна точност). Тя заема по – голяма

компютърна памет, но има интервал от 5.0E-324 до 1.7E308 и висока точност от 15-16 знака. Ако пък искате максимална прецизност, но изисквате и малко по-малък обхват, може да използвате типа **decimal**. Това използва два пъти по-голямо място в паметта от **double** и съдържа стойности в обхвата на 28-29 знака.

<i>float</i>	7 знака	$\pm 1.5 \times 10^{-45}$ до $\pm 3.4 \times 10^{38}$
<i>double</i>	15-16 знака	$\pm 5.0 \times 10^{-324}$ до $\pm 1.7 \times 10^{308}$
<i>decimal</i>	28-29 знака	$\pm 1.0 \times 10^{-28}$ до $\pm 7.9 \times 10^{28}$

Също така може да използвате експонента, за да изразите **double** и **float** стойности. Например:

9.4605284E15

Това е стойност на число с плаваща запетая и двойна точност, което всъщност представлява метрите, които светлината изминава за период от една година.

Когато промените стойност на променлива с плаваща запетая и двойна точност (**double**) в обикновена стойност с плаваща запетая (**float**), част от прецизността бива загубена. Това означава, че трябва да предприемете специални мерки, за да бъдете сигурни като програмист, че искате точно това да се случи и че можете да живеете с последиците. Този процес е известен като **кастване**²¹.

²¹Каждането позволява копиране на стойност от променлива от един тип данни в друг. Термин от Английски език = **cast, casting**, вж.
<http://msdn.microsoft.com/en-us/library/ms173105.aspx>

Гледна точка на програмиста:

Простите променливи са най – вероятно и най – добрите. Повечето програмисти са склонни да ползват по-чести цели числа (*int*) и числа с плаваща запетая (*float*) за типове на променливи. Това прави програмата по – лесна за разбиране.

Char е тип променлива, която може да съдържа единичен символ. Символ е това, което получаваме, когато натиснем копче на клавиатурата или изведем единичен символ на екрана. C# използва комплект от символи наречен *Unicode*, който може да борава с над 65000 различни символни знаци.

Escape последователност

Escape последователността е специална комбинация от символи, които винаги започват със специалния избягващ символ за обратно наклонена черта. По този начин се отменя значението на следващия знак и всъщност комбинацията предоставя нова функционалност. Възможни *Escape* последователности са:

последователност	Български език	Английски език
<code>\'</code>	Единична кавичка	<i>Single quote</i>
<code>\"</code>	Двойна кавичка	<i>Double quote</i>
<code>\\</code>	Обратно наклонена черта	<i>Backslash</i>
<code>\0</code>	Нула	<i>Null</i>
<code>\a</code>	Сигнал за тревога	<i>Alert</i>
<code>\b</code>	Връщане назад	<i>Backspace</i>
<code>\n</code>	Нова линия	<i>New line</i>
<code>\r</code>	Връщане на каретката	<i>Carriage return</i>
<code>\t</code>	Хоризонтална табулация	<i>Horizontal tab</i>
<code>\v</code>	Вертикална кавичка	<i>Vertical quote</i>

Кодова стойност на символите

Ако искате можете да изразите стойност на символ, чрез цифровата му стойност от таблицата с *Unicode* знаци. Например десетичният цифров еквивалент на главната буква „А“ е 65. Това представяне в шестнадесетичната бройна система (с база 16) е четири по шестнадесет и една единица по шестнадесет (*тоест 41*). Следователно може да бъде записано в програмата като:

```
char A = '\x0041';
```

Забележете, че трябва да поставим две нули пред двете шестнадесетични цифри.

Низови променливи

Тип кутия, която може да съдържа низ от текст. Променливата *string* може да съдържа ред от текст:

```
string text = "това е просто един текстов низ";
```

Ако просто искаме да изразим текст без *Escape* последователност, можем да кажем на компилатора, че това е дословен низ, като в този случай поставяме @ пред него:

```
Console.WriteLine(@"\x0041BCDE\a");
```

в резултат на екрана ще се отпечата:

```
\x0041BCDE\a
```

Посоченото е полезно при изразяване на пътища към файлове.

Запазване на състояние чрез bool

Променлива от тип *bool* е тип кутийка, която съдържа информация дали нещо е вярно или е невярно (т.е. *истина* или *лъжа*). Няма нужда да губите излишно пространство от паметта, чрез използване на друг тип, който може да побере голям брой възможни стойности. Когато просто трябва да се съхранява дали изразът е вярно или невярно, в този случай това са само две

стойности, които типа *bool* позволява. Например:

```
bool networkOK=true;
```

Идентификатори

В C# идентификаторите са имената, които програмиста използва в своята програмата, или по – правилно имената на променливите се наричат идентификатори. В C# има някои правила, за това какво представлява правилен идентификатор:

- Всички имена на идентификатори трябва да започват с буква.
- След буквата можем да имаме или буква или цифра или символа долната черта „_“.
- Главните и малки букви са различни, тоест *Митко* и *митко* са различни идентификатори.

Задаване на стойности на променливите

Веднъж след като сме създали променлива, трябва да знаем как да и поставим нещо в нея и как да изведем стойността ѝ. C# извършва това чрез присвояване. Присвояването има две части: първата част е онова, което искате да бъде присвоено, а втората част е мястото, където искате да го поставите. Например нека разгледаме следната програмна конструкция за присвояване:

```
class Homework
{
    static void Main()
    {
        int first, second, third;
        first=1;
        second=2;
        third= first+second;
    }
}
```

Операцията: „Вземи резултата от дясната страна на присвояването и го пусни в кутията отляво“, може да изглежда

безобидно, но програмна конструкция за присвояване от следния вид:

```
2=second+1;
```

е част от програмна проклетия, която би довела до появата на неприятни грешки.

Изрази

Изразът може да бъде пресметнат, така че да се получи краен резултат. Използваме този резултат, както намерим за добре в нашата програма. Изразите могат да бъдат прости – състоящи се в пресмятане на единична стойност и сложни – състоящи се от поредица големи изчисления. Те винаги са съставени от - оператори и операнди.

Операнди

Операндите са елементи, с които операторите работят. Обикновено те са точни стойности или идентификатори на променливи.

Оператори

Операторите вършат работата. Те уточняват кои операции да бъдат извършени върху операндите. Например изразът:

```
2+3*4-1+3*(2+3)*4
```

се обработва (*пресмята*) от C# движейки се от ляво надясно. C# прави това, като дава приоритет на всеки оператор. Когато C# обработва израз, преглежда за оператори с най-висок приоритет като ги извършва първи, след което извършва операторите с по-нисък приоритет.

За пълнота ето лист от всички оператори и какво те правят. Ще изброим операторите по приоритет в таблица по-долу, като

операцията с най-висок приоритет е в началото на таблицата, а тази с най – нисък в края.

-	унарен минус	минусът, с който C# намира отрицателните числа, например (-1) унарен минус означава, че се прилага само върху един операнд.
*	мултипликация	забележете, че се използва *, вместо по – математически вярното.
/	деление	заради трудността да нарисуваме едно число над друго използваме този символ вместо това.
+	събиране	аналог на математическата операция.
-	изваждане	забележете, че използваме абсолютно същия знак като унарния минус.

Понеже тези оператори работят с числа, те често се наричат **числени оператори**.

Промяна на типа на данните, чрез разширяване и стесняване

За да разберете какво всъщност имам предвид да разгледаме пример с куфари. Като се приготвям за пътуване обикновено взимам куфар. Ако реша да сместя багажа в по-малък куфар, ще трябва да извадя всичко от големия куфар и да го сложа в по-малкия. Но мястото може да не стигне и ще се наложи да оставя някоя от дрехите си. Това се нарича „стесняване“. Ако обаче взема по-голям куфар, няма да има проблем. Големият куфар ще поеме всичко, което е било в по – малкия и ще има място за още, т.е. ако напиша:

```
int i=1;  
float x=i;
```

Това ще работи добре, защото типа с плаваща запетая **float** може да съдържа всички стойности поддържани от типа цяло число.

Обаче, ако напиша:

```
float x=1;  
int i=x;
```

това ще накара компилаторът да се оплаче (*въпреки, че в момента променливата x съдържа цяла стойност*). Компилаторът е загрижен, че може да загубим информация, чрез такова присвояване и третира това като грешка.

Кастване

Можем да накараме C# да разгледа стойност все едно е от определен тип, чрез използването на **кастване**²². Кастваме стойност като напишем типа, който искаме да получим в скоби преди нея. Например:

```
double d=1.5;  
float f=(float)d;
```

В горния код посланието към компилатора е: *„Не ме интересува, че присвояването може да причини загуба на информация. Аз, като автор на програмата, ще поема отговорността програмата да работи правилно.“*

Кастване също така може да причини загуба на информация:

```
int i;  
i=(int)1.999;
```

Този процес изхвърля дробната част, което означава, че накрая променливата ще има стойност единица в себе си.

Кастване и буквални стойности

```
float x;  
x=3.4;
```

²²Кастването позволява копиране на стойност от променлива от един тип данни в друг. Термин на Английски език = **cast, casting**, вж.

[http://msdn.microsoft.com/en-US/library/ms173105\(v=vs.110\).aspx](http://msdn.microsoft.com/en-US/library/ms173105(v=vs.110).aspx)

Този код изглежда абсолютно правилен. Обаче не е. Това е така, защото буквалната стойност на 3.4 е стойност с двойна точност тип **double**, когато бъде изразена като буквална, а променливата x е декларирана като плаваща запетая. Ако искаме да сложим буквална стойност с плаваща запетая в променлива с плаваща запетая тип **float**, можем да използваме кастване. Например:

```
float x;  
x=(float)3.4;
```

Това каства буквалната стойност с двойна точност в стойност с плаваща запетая, така че присвояването работи. За да ни направят живота по-лесен, създателите на C# са добавили различни начини, по които да изразим буквална стойност с плаваща запетая в програма. Ако поставите f след стойността това ще се зачита като стойност с плаваща запетая. Това означава, че можем да запишем израза и по този начин:

```
float x=3.4f;
```

Типове данни в изразите

Когато C# използва оператор, той прави решение какъв тип резултат ще бъде получен. По същество, ако двата операнда са цели числа, то резултата ще бъде цяло число. Ако те са с плаваща запетая, то резултата трябва също да е плаваща запетая. Това може да доведе до проблеми:

```
1/2  
1/2.0
```

Може да си мислите, че тези изрази ще дадат еднакъв резултат. Да, ама не. За първия израз, който съдържа само цели числа, компилатора ще произведе целочислен резултат. Следователно ще пресметне цялата стойност, която е 0 (*дробната част винаги се отсича*). Понеже вторият израз съдържа стойност с плаваща запетая, то компилатора ще върне стойност с плаваща запетая

като резултат, което е по-точния отговор, а именно 0.5.

Ако искате пълен контрол над определен вид оператор, който компилатора да генерира, програмата трябва да съдържа изрични каствания, за да установи правилния контекст за оператора.

```
using System;

class CastDemo
{
    static void Main()
    {
        int i=3, j=2;
        float fraction;
        fraction=(float)i/(float)j;
        Console.WriteLine("fraction: "+fraction);
    }
}
```

Гледна точка на програмиста:

Кастването може да добави яснота. Склонен съм да слагам каствания дори и да не са нужни, това може да направи програмата по – ясна.

Писане на програма

Всъщност програмите, които пишем са по-сложни. Например, може да се наложи програмата Ви да вземе решение, въз основа на данни, които са дадени. Те повтарят определени операции многократно, докато нещо е вярно. Друг пример е прочитане на голямо количество данни и тяхната обработка след това, по редица различни начини. Тук ще разгледаме как да дадем на нашите програми, допълнително ниво на сложност.

Блокови коментари

/ Тази програма намира количеството стъкло и количеството дърво необходими за създаване на един*

прозорец */

Линейни коментари

Друга форма на коментиране използва поредицата `//` . Това маркира началото на коментар, който е разположен до края на програмния ред.

```
// придвижваме се към следващия елемент  
position = position + 1;
```

Контролиране на програмния поток

Условен оператор

```
if (условие)  
    // израз или блок, който изпълняваме, ако условието е  
    вярно  
else  
    // израз или блок, който изпълняваме, ако условието е
```

Чрез оператори за сравнение можем да създадем условия, които връщат логически резултати, наричат се **логически състояния**. Те имат две възможни стойности: вярно или грешно, истина или лъжа, **true** или **false**, 1 или 0.

```
if (true)  
    Console.WriteLine ("Здравей!");
```

Състояния и оператори за сравнение

`==`

е равно на. Ако лявата страна и дясната страна на оператора са равни, то изразът ще върне стойността *вярно*. Ако не са равни, то върнатата стойност е *грешно*.

`!=`

не е равно на. Обратното на равенството.

`<`

по-малко от. Ако операнда от лявата страна е по-малък от този

от дясната страна, стойността на израза е *вярно*. Ако левият оператор е по-голям или равен от десния, то изразът връща *грешно*.

>

по-голямо от. Ако операнда от ляво е по-голям от този от дясно, то резултата е *вярно*. Ако операнда от ляво е по-малък или равен на този в дясно, то резултата е *грешно*.

<=

по-малко или равно. Ако операнда от ляво е по-малък или равен на този от дясно получаваме *вярно*, иначе получаваме *грешно*.

>=

по-голямо или равно. Ако операнда от ляво е по-голям от или равен на този от дясно получаваме *вярно*, иначе – *грешно*.

!

НЕ. Това може да се използва за преобръщане определена стойност или израз, например можем да кажем *!true*, което всъщност е *false*.

Комбиниране на логически оператори

&&

И. Ако операндите на всяка страна на && са верни, резултата на && е *вярно*. Ако един от тях е грешен, то резултата е *грешно*.

||

ИЛИ. Ако, която и да е операнда на всяка страна на || е вярна, резултата от израза е *вярно*. Изразът е *грешно* само, ако и двете операнди са грешни.

Използвайки тези оператори в съчетание с конструкцията на *if*, можем да направим решение и да променим това, което нашата програма ще прави, в съответствие на данните, които получаваме.

Използване на блокове за комбиниране на изрази

Няколко програмни конструкции скупчени заедно между символите { и } се разглеждат като цяла програмна конструкция, така че можем да направим следното:

```
if(width>5.0)
{
    Console.WriteLine("широчината е твърде голяма.Използвам
максималната.\n");
    width=5.0;
}
```

Тези две програмни конструкции вече са блок, който ще бъде изпълнен само, ако широчината е по-голяма от 5.0.

Цикли

Условните конструкции ни позволяват да изпълняваме определени програмни конструкции или блокове, еднократно и то само ако даден израз е верен. Практически често се налага да изпълняваме програмна конструкция или блок, многократно, докато определен израз е верен. Многократно изпълнение в този контекст означава повторно изпълнение на същата програмна конструкция или блок, краен брой пъти. Програмните конструкции, които позволяват многократно изпълнение на код се наричат **цикли**.

цикъл do ... while

```
do
    // програмна конструкция или блок
while ( условие );
```

Това ни позволява да повтаряме програмна конструкция или

блок, докато условието в края на цикъла стане грешно.

```
using System;
class ForeverExample
{
    public static void Main ()
    {
        do
            Console.WriteLine("Здравей!");
        while(true);
    }
}
```

цикъл while

while (условие)
// програмна конструкция или блок

Понякога искаме да решим дали да повторим цикъла или не преди да го изпълним.

```
using System;
class whileExample
{
    public static void Main()
    {
        int i;
        i=1;
        while(i<11)
        {
            Console.WriteLine("Здравей!");
            i=i+1;
        }
    }
}
```

цикъл for

for(начална стойност;
 проверка на стойността за изход;
 изменение на стойността или стъпка)
{
 // програмни конструкции, изпълняващ се даден брой
 пъти
}

Често искаме да повторим програмни конструкции даден брой пъти.

```

using System;

class ForExample
{
    public static void Main()
    {
        int i;
        for(i=1;i<11;i=i+1)
        {
            Console.WriteLine("Здравей!");
        }
    }
}

```

Началната стойност се записва в контролна променлива, с която започва цикъла. Проверката на стойността за изход е условие, което трябва да е вярно за да продължи цикъла *for*. Изменението на стойността или стъпка е действие, което се изпълнява, за да се актуализира контролната променлива в началото на всеки кръг при изпълнението.

Прекъсване на цикъл

Понякога искаме да прекъснем цикъл, докато сме по средата му. Правим това посредством програмната конструкция ***break***. Това е команда за излизане веднага от цикъл.

```

while (всичко е наред)
{
    // сложни работи
    ....
    if (прекъснат)
    {
        break;
    }
    ....
    // още сложни работи
    ....
}
// тук стигаме, ако прекъсването стане вярно

```

Можем да прекъснем всеки от трите вида цикли. Във всеки случай, програмата продължава да работи с последната програмна конструкция от цикъла.

Връщане обратно в началото на цикъл

Понякога ще искаме да се върнем в началото на цикъл и да го изпълним отново. Това се случва, когато сме изпълнили толкова програмни конструкции, колкото искаме. C# ни позволява да използваме ключовата дума **continue**, например:

```
for(step=1; step<counter; step=step+1)
{
    ....
    // програмни конструкции
    ....
    if (свършили сме всичко, което искаме) continue;
    ....
    // още програмни конструкции
    ....
}
```

Кратки оператори

Операторът ++ е наречен **унарен оператор**, защото работи само с една операнда. Той кара стойността в тази операнда да бъде увеличена с единица (*действие наричано още: инкрементиране*). Има съответстващ обратен оператор --, който може да бъде използван, за да намали стойността на променлива с единица (*действие наричано още: декрементиране*).

$a += b$	стойността в a е заменена с $a + b$
$a -= b$	стойността в a е заменена с $a - b$
$a /= b$	стойността в a е заменена с a / b
$a *= b$	стойността в a е заменена с $a * b$

Други кратки оператори

Когато разглеждаме оператори, като ++ има вероятност за неяснота, в смисъл, че не знаем дали вземаме стойността преди или след инкрементацията.

i++

Означава: „Дай ми стойността преди инкрементацията.“

++i

Означава: „Дай ми стойността след инкрементацията.“

```
int i=2, j;  
j=++i;
```

Спретнато отпечатване

Използвайте контейнери, които просто маркират мястото, където стойността ще се отпечата. Подобен контейнер, който се замества с конкретна стойност е показан в примера по-долу:

```
int i=150;  
double f=1234.56789;  
Console.WriteLine("i: {0} f: {1}",i,f);  
Console.WriteLine("i: {1} f: {0}",f,i);
```

В резултат от този програмен код на екрана ще се отпечата:

```
i: 150 f: 1234.56789  
i: 150 f: 1234.56789
```

Настройване на точността на реално число

Контейнерите могат да имат добавена допълнителна информация за форматирането:

```
int i=150;  
double f=1234.56789;  
Console.WriteLine("i: {0:0} f: {1:0.00}",i,f);
```

Това ще отпечата на екрана:

```
i: 150 f: 1234.57
```

Уточняване на броя на отпечатани знаци

Можем да уточним броя на отпечатваните знаци, като ги фиксираме със съответния брой нули:

```
int i=150;
double f=1234.56789;
Console.WriteLine("i: {0:0000} f: {1:00000.00}",i,f);
```

Това ще отпечата на екрана:

```
i: 0150 f: 01234.57
```

Наистина капризно форматиране

Ако искаме истински капризно форматиране, можем да използваме символът #. Символът # във форматния низ означава „*сложи знак тук, ако има такъв*“:

```
int i=150;
double f=1234.56789;
Console.WriteLine ("i: {0:#,##0} f: {1:##,##0.00}",i,f);
```

Използвахме символа #, за да изведем хилядите разделени със запетая:

```
i: 150 f: 1,234.57
```

Отпечатване в колони

Не на последно място, можем да добавим стойност за широчина към информацията за оформление на отпечатването. Това е много полезно, ако искаме да печатаме в колони, например:

```
int i=150;
double f=1234.56789;
Console.WriteLine("i: {0,10:0} f: {1,15:0.00}",i,f);
Console.WriteLine("i: {0,10:0} f: {1,15:0.00}",0,0);
```

Това ще отпечатана екрана следния изход:

```
i:           150 f:           1234.57
i:              0 f:              0.00
```

Стойността на числото **int** е отпечатано в колона с големина 10 символа, а **double** променливата е отпечатана в колона с големина 15 символа. В момента на отпечатване изхода е подравнен в дясно, ако искахме да бъде подравненв ляво,

бихме направили широчината отрицателна:

```
int i=150;  
double f=1234.56789;  
Console.WriteLine("i: {0,-10:0} f: {1,-15:0.00}",i,f);  
Console.WriteLine("i: {0,-10:0} f: {1,-15:0.00}",0,0);
```

Този програмен фрагмент, ще изведе на екрана:

```
i: 150 f: 1234.57 i: 0 f: 0.00
```

ГЛАВА 3. СЪЗДАВАНЕ НА ПРОГРАМИ

Настоящата глава продължава да развива програмните умения: запознавайки с необходимостта от методи, техните параметри и върнати стойности; разделяне на програмата на управляеми парчета; манипулиране на голям обем данни използвайки масиви; обработка на грешки посредством хващане и хвърляне на изключение; използване на файлове, чрез изходни потоци за запис във файл и входни потоци за четене от файл.

Необходимостта от методите

До сега всички наши програми използваха само един метод. Както знаем методът е блок от код, който следва главната част на нашата програма. C# обаче ни позволява да създадем и други методи, които могат да бъдат използвани, когато програмата ни работи. Методите ни дават две нови оръжия:

- Можем да напишем методи, за да използваме многократно, дадена част от кода, който сме написали.
- Можем също така да използваме методи, за да разделяме големи задачи за решаване на няколко по-малки.

В основата си при създаването на метод, можем да вземем блок от код и да му дадем име. След това можем да извикаме този блок от код, който ще свърши определената работа за нас.

```
using System;

class MethodDemo
{
    static void Print()
    {
        Console.WriteLine("Един ред отпечатан текст.");
    }

    public static void Main()
    {
        Print();
        Print();
    }
}
```

По този начин можем да използваме методите, за да ни предпазват от писането на един и същи код два пъти. Просто поставяме кода в тялото на метод и го извикваме, тогава когато е необходимо.

Параметри

Параметър е средството, което използваме, за да подадем стойност при извикването на даден метод. Данните се подават на метода, с цел да бъдат обработени от него.

```
using System;

class MethodParametherDemo
{
    static void PrintValue(int i)
    {
        Console.WriteLine("Стойността на променливата е "+i);
    }

    public static void Main()
    {
        PrintValue(101);
        PrintValue(500);
    }
}
```

В блока от код, който наричаме тялото на този метод, може да използваме параметъра *i* като променлива от тип цяло число. Когато методът бъде стартиран, стойността, която се доставя за параметъра се копира в метода и по-точно в параметъра *i*.

Върнати стойности

Методът също така може да върне стойност, когато е повикан. Вече използвахме това свойство, например и двата метода *ReadLine* и *Parse* връщат резултат, който сме използвали в нашите програми.

```

using System;

class MethodReturnValueDemo
{
    static int PrintValuePlusOne(int i)
    {
        i = i+1;
        Console.WriteLine("Стойността на променливата е "+i);
        return i;
    }

    public static void Main ()
    {
        int result;
        result = PrintValuePlusOne(5);
        Console.WriteLine("Стойността сега е "+result);
    }
}

```

Методът *PrintValuePlusOne* взема стойността на параметъра и го връща обратно увеличен с единица.

Полезен метод

```

static double EnterValue
(
    string information, // информация с подсказка за
    потребителя
    double min,         // най – малката позволена стойност
    max                 // най – голямата позволена стойност
)
// начало на блока на метода
{
    double result = 0;
    do {
        Console.WriteLine(information+" между "+min+" и
"+max);
        string resultText = Console.ReadLine();
        result = double.Parse(resultText);
    } while ((result<min)|| (result>max));
}

```

Методът *EnterValue* задължава потребителят да въведе стойност, която е между най – малка и най – голяма зададена предварително стойност. По-късно той може да бъде използван, за да прочита стойностите и да проверява, че те са в зададения обхват. Например:

```

double width = EnterValue("Въведи дължина на прозореца:
",1.5,3.5);

```

Гледна точка на програмиста: Проектирайте с методи

Често се случва, докато пишете код да повтаряте дадено действие. Ако правите това, ще е добра идея да вземете посоченото действие и да го преместите в метод. Има две причини, защо това е добра идея:

1: Спасява ви от писането на един и същи код два пъти.

2: Ако е намерена грешка в кода трябва да я поправите само един път.

Преместване на кода в програмата и създаване на методи се нарича **refactoring**. То е важна част от Софтуерното инженерство, което ще направим по – късно.

Подаване на параметър чрез стойност

```
static void ParameterAddOne(int i)
{
    i = i+1;
    Console.WriteLine("Стойността на променливата е "+i);
}
```

Методът *ParameterAddOne* добавя едно към параметъра; отпечатва резултата и тогава връща резултата.

```
int test=20;
ParameterAddOne(test);
Console.WriteLine("Стойността на тестовата променлива е "+test);
```

Частта от C# по - горе извиква метода с променливата *test* като параметър. Когато метода бъде извикан, той отпечатва на екрана на компютъра следното:

```
Стойността на променливата е 21
Стойността на тестовата променлива е 20
```

Подаването на параметър чрез стойност е доста безопасно, защото каквото и да направи метода, той не може да засегне стойността на променливата от кода, който извиква метода.

Подаване на параметри чрез референция

В методите често вместо използване на стойност от променлива

се използва референция, за да се вземе същинската променлива. Ефективно, това което се подава в метода е позицията или адреса на променливата в паметта, вместо съдържанието ѝ.

Ако подавате параметър чрез референция, промените на параметъра променят също така и стойността на променливата, чиято референция сте подали.

```
static void ReferenceAddOne(ref int i)
{
    i = i+1;
    Console.WriteLine("Стойността на променливата е "+i);
}
```

Забележете, че ключовата дума *ref* е добавена като информация за параметъра.

```
test = 20;
ReferenceAddOne(ref test);
Console.WriteLine("Стойността на тестовата променлива е "+test);
```

В горният код се прави извикване на метод и също така има думата *ref* пред параметъра. В този случай крайния резултат ще бъде:

```
Стойността на променливата е 21
Стойността на тестовата променлива е 21
```

Полезен съвет: Документирайте страничните ефекти

С други думи, когато някой извика нашия метод „ReferenceAddOne“ ще има „страничния ефект“ да промени нещо извън самия метод (а именно стойността на параметъра, подадена чрез референция). В общи линии трябва да бъдете внимателни със страничните ефекти, в смисъл, че този, който чете Вашата програма трябва да знае, че Вашият метод прави промени по този начин.

Подаване на параметри като изходящи референции
Искаме да прочетем името и годините на потребителя.

Оригиналната стойност на параметрите не представлява интерес за метода. Вместо това той просто иска да достави резултати до тях. В този случай можем да сменим ключовата дума *ref* с *out*:

```
static void EnterPerson(out string name, out int age)
{
    name = EnterText("Въведи твоето име: ");
    age = EnterNumber("Въведи твоята възраст: ",0,100);
}
```

Методът *EnterPerson*, чете името и годините на един човек. Забележете, че използва още два метода, които трябва да сте създали преди това *EnterText* и *EnterNumber*. Можем да извикаме *EnterPerson* по следния начин:

```
string name;
int age;
EnterPerson(out name, out age);
```

Забележете, че трябва да използваме ключовата дума *out* също и при извикването на метода.

Библиотеки от методи

```
static string EnterText(string info)
{
    string result;
    do {
        Console.Write(info);
        result = Console.ReadLine();
    } while (result=="");
    return result;
}

static int EnterNumber(string info,int min,int max)
{
    int result;
    do {
        string numberText= EnterText(info);
        резултат = int.Parse(numberText);
    } while ((result<min)|| (result>max));
    return result;
}
```

Можем да използваме методите по следния начин:

```
string name = EnterText("Въведи твоето име: ");  
int age = EnterNumber("Въведи твоята възраст: ", 0, 100);
```

Всъщност онова, което съм склонен да правя, когато работя по конкретен проект е да създам малки библиотеки от полезни методи, като тези в примера по - горе, които мога да използвам.

Полезен съвет:

*Винаги вземайте под внимание възможността за провал (**failure behaviours**). Когато пишете един метод, винаги мислете за начините, по които той може да се провали.*

Ако при изпълнение на метода възникне грешка, то трябва напишем функционалност, която да доставя състоянието на грешката към кода, който извиква метода. Често този проблем може да бъде разрешен, като пишем методите, така че да връщат стойност. Ако върнатата стойност е 0, това означава, че методът се е изпълнил коректно и е върнал правилна стойност. Ако върнатата стойност не е нула, това означава, че методът не е проработил и върнатата стойност е грешка, която посочва какво точно се е объркало.

Обхват на променливите

Обхват и блокове

Обхвата на локална променлива е в рамките блок от код, в който променливата е декларирана. Можем да декларираме променлива по всяко време в блока, но задължително тя трябва да бъде декларирана преди да я използваме. Когато изпълнението на програмата премине извън блок, всички локални променливи, които са декларирани в този блок автоматично се изхвърлят (*т.е. унищожават*).

Локални променливи за цикъла *for*

Специален вид променливи могат да бъдат използвани, когато създаваме конструкцията на цикъла *for*. Това ни позволява да декларираме контролна променлива, която да съществува в рамките на продължителността на цикъла *for*.

```
for(int i=0;i<10;i=i+1)
{
    Console.WriteLine("Здрасти!");
}
```

Член данни в класове

Ако искаме да позволим на два метода в един и същи клас да споделят променлива ще трябва да направим променливата член на този клас. Това означава да я декларираме извън методите в класа:

```
class ClassVariableDemo
{
    // Променливата е част от класа
    static int value = 0;

    static void SetValue99()
    {
        стойност = 99;
    }

    static void Main()
    {
        Console.WriteLine("Променливата има стойност:
"+value);
        SetValue99();
        Console.WriteLine("Новата стойност вече е: "+value);
    }
}
```

Променливата е сега част от класа *ClassVariableDemo*, така че както метод *Main*, така и метод *SetValue99*, могат да я използват. Горната програма ще отпечата:

```
Променливата има стойност: 0
Новата стойност вече е: 99
```

Променливите в класове са много полезни, ако искаме да имаме няколко метода „споделящи“ определени данни. Маркирането

на променлива като *static* означава „променливата е част от клас и е винаги присъстваща“.

Полезен съвет: Планирайте използването на променливите

Трябва да решите кои променливи са нужни само за локални блокове и кои трябва да бъдат член данни на даден клас.

Гледайте броя на променливите, които са член данни в даден клас, да бъде минимален. Използвайте локални променливи само, ако имате нужда да запазите стойност в променлива за малка част от кода.

Масиви

Изглежда вече знаете почти всички особености на езика, които са нужни за реализиране на всяка програма, която някога е била писана. Само още едно нещо липсва и това е способността да създавате програми, които запазват голям обем от данни. Масивите са един начин да направим това.

Масивът²³ ни позволява да декларираме цял ред от определен вид кутийки. Можем да използваме след това **индекси**²⁴, за да посочим коя кутийка в реда искаме да използваме.

Да разгледаме следният пример:

²³Масив е колекция от елементи от еднакъв тип, като стойностите на елементите се достъпват по индекс. Термин на Английски език = **Array**, вж.

<http://en.wikipedia.org/wiki/Array> и <http://msdn.microsoft.com/en-us/library/vstudio/9b9dty7d.aspx>

²⁴Индекс е номерация на елементите в масива, обикновено започваща от нула. Термин на Английски език = **Index**

```

using System;

class ArrayDemo
{
    public static void Main()
    {
        int [] points = new int[11];
        for (int i=0;i<11;i=i+1)
        {
            points[i] = EnterNumber("Точки: ",0,1000);
        }
    }
}

```

Частта `int [] points`, казва на компилатора, че искаме да използваме масив. Частта, която създава самия масив е `new int [11]`. Когато C# види това, той си казва: „Аха! Това, което ни трябва тук е масив.“. След това взема няколко парчета дърво и прави дълга тънка кутия с 11 отделения в нея, всяко едно от тях е достатъчно голямо да съдържа единично цяло число `int`.

Елементи на масива

Всяко отделение в кутията се нарича елемент. В програмата може да определим кой елемент имаме впредвид, като сложим неговият номер в квадратни скоби `[]` след името на масива. Тази част се нарича **индекс**.

Номериране на елементите в масива

C# номерира кутийките започвайки от 0. Ако погледнете частта от програмата, която чете стойностите в масива ще видите, че броим само от 0 до 10. Това е много важно! При опит да отидем извън границите на масива *точки* това ще доведе до грешка, когато програмата бъде стартирана.

Големи масиви

Добър трик, когато работим с масиви, е да използваме константни променливи, които да пазят големината на масива. Това има две значителни ползи:

- Прави програмата лесна за разбиране;

- Прави програмата лесна за променяне.

Стойността на константна променлива се задава, когато тя бъде декларирана.

```
using System;

class ArrayDemo2
{
    public static void Main ()
    {
        const int MAX_POINTS = 1000;
        int [] points = new int [MAX_POINTS];
        for (int i=0;i<MAX_POINTS;i=i+1)
        {
            points[i] = EnterNumver("Точки: ",0,1000);
        }
    }
}
```

Има **установена практика**²⁵ константите от този вид да се пишат с ГОЛЕМИ БУКВИ и с долна черта между думите. Навсякъде преди това са използвани фиксирани променливи, които да обозначават размера на масива вместо това сега се използва константа.

Създаване на двумерен масив

Понякога искаме **мрежа**²⁶. Можем да направим това използвайки двумерен масив:

```
int [,] matrix = new int [3,3];
matrix[1,1] = 1;
```

Това изглежда почти като нашия едномерен масив, но има някои важни разлики. Сега между квадратните скоби има запетая [,].

²⁵Установената практика в средите на програмистите се нарича с по популярното име: Конвенция, идващо от термин на Английски език = **Convention**

²⁶Двумерния, двуизмерен или двуменсионен масив (*име което идва от броя на измеренията или дименсиите му*), често се нарича матрица или мрежа, на Английски език = **matrix, grid**

	0	1	2
0	0	0	0
1	0	1	0
2	0	0	0

Първият индекс посочва колона, а вторият индекс посочва ред в масива.

Повече от две дименсии

Веднъж на **100**²⁷ години, може да ви се наложи да използвате повече от две измерения.

```
int [,] cube = new int [3,3,3];
cube[1,1,1] = 1;
```

Този код създава три измерна дъска и след това записва единица точно в средата на игралния куб.

Забележете, че понякога може да си мислите, че трябва да добавите още една дименсия, когато това, от което всъщност се нуждаете е да добавите друг масив, например:

```
int [] points = new int [11];
string [] names = new string [11];
```

В този случай Вашата програма ще трябва да се погрижи информацията за името и точките на всеки играч да са винаги свързани заедно. С други думи елемента с индекс 0 в масива *имена* трябва да съдържа името на играча направил точки, записани в елемента с индекс 0 в масива *точки*.

Изключения и грешки

Ако има едно нещо, което научих от програмирането е това, че когато някой направи така, че Вашата програма да се провали, то

²⁷Веднъж на 100 години или когато изгрее синята луна (*т.е. в много редки случаи*). В превод от Английски език = **Once in a blue moon**

той изглежда умният, а вие програмиста изглеждате глупавият. Вижте следния пример:

Въведете вашата възраст:
Двадесет и едно

Потребителят въвежда неговата възраст, но той не използва числа. Някой е бил достатъчно „умен“ да напише неговата възраст с думи и по този начин той ще може да счупи програмата.

Parse е методът, който използваме, за да превръщаме низове от текст в числени стойности.

```
int age = int.Parse(ageText);
```

Проблемът с *Parse* е, че ако му дадем низ, който съдържа невалиден текст, той няма да знае какво да прави с него. *Parse* решава своите проблеми, като **хвърля**²⁸ **изключение**²⁹ към друга част на програмата, която да **хване**³⁰ и обработи събитието. Ако нищо не хване изключението, тогава изключението ще приключи програмата. С# системата ще изведе съдържанието на грешката, когато стигне до нея и ще спре.

Хващане на изключения

Можем да накараме нашата програма да се справя с невалиден текстов вход към *Parse*, като добавим нов код, който ще хваща изключенията, които *Parse* хвърля и ще се опитва да оправи

²⁸Термин на Английски език = **throw, throwing**, вж.

<http://msdn.microsoft.com/en-us/library/1ah5wsex.aspx>

²⁹Изключение е процес на отговор в резултат на извънредни събития, които възникват по време на изпълнение на програмата и които изискват специална обработка. Термин на Английски език = **exception**, вж.

http://en.wikipedia.org/wiki/Exception_handling

³⁰Термин на Английски език = **catch, catching**, вж. <http://msdn.microsoft.com/en-us/library/vstudio/0yd65esw.aspx>

проблема. На програмен език това се нарича „*динамично справяне с грешки*“³¹, смисълът тук е, че нашата програма ще отговаря на грешки, когато те се случат.

```
int age;
try
{
    age = int.Parse(ageText);
    Console.WriteLine("Благодаря!");
}
catch
{
    Console.WriteLine("Грешна стойност за възраст!");
}
```

Кодът по-горе използва метода *Parse* за да декодира низа *ageText*. Обаче действието на метода *Parse* се случва в блокът *try*. Ако извикването на *Parse* хвърли изключение, ще се изпълни кода в хващащия блок *catch* и това ще изведе съобщение за грешка към потребителя.

Обекта изключение

Изключението е тип обект, който съдържа детайлна информация за възникналия проблем. Когато *Parse* се провали той създава обект от тип изключение, в който е описано лошото нещо, което току що се е случило (*в този конкретен случай входния низ не е в правилния формат*). Горната програма игнорира изключението и просто регистрира събитието изключение, но ние можем да подобрим диагностиката на нашата програма, като прихванем изключението, по следния начин:

³¹ „Динамично справяне с грешки“ е израз в превод от Английски език = **dynamic error handling**

```

int age;
try
{
    age = int.Parse(ageText);
    Console.WriteLine("Благодаря!");
}
catch (Exception e)
{
    // съобщението за грешка при изключение
    Console.WriteLine(e.Message);
}

```

Типа изключение има свойство наречено съобщение, което съдържа низ, описващ възникналата грешка. Ако потребителят напише невалиден низ горната програма ще напише текста в изключението, а именно:

```
Input string was not in a correct format.
```

Кое то в превод на Български език гласи: *„Въведения низ не беше в правилния формат.“*. Това е много полезно, ако кода в блока на *try* хвърли няколко различни вида изключения.

Добавяне на клаузата *finally*

Понякога има програмни конструкции, които програмата трябва да направи независимо дали е възникнало изключение или не. Програмните конструкции в клаузата ***finally*** ще се извършат независимо дали програмата в блока на *try* хвърли изключение.

```

try
{
    // Код, който потенциално може да хвърли изключение
}
catch (Exception outer)
{
    // Код, който хваща и обработва изключението
}
finally
{
    // Това се изпълнява независимо дали има изключение
    или не
}

```

В горният примерен код изразите в частта ***finally*** е гарантирано, че ще бъдат изпълнени, или след приключване на изпълнението

на изразите от частта **try** или точно след изпълнението на изразите от частта **catch** на Вашата програма.

Хвърляне на изключение

Сега след като знаем как да хващаме изключения, следва да вземем под внимание как да ги хвърляме изключения. Всъщност това е много лесно:

```
throw new Exception("Бум!");
```

Хвърляне на изключение може да накара Вашата програма да приключи своята работа, ако кода не е изпълнен в **try ... catch** конструкция. Ако искате да хвърлите изключение това трябва да е в такава ситуация, в която Вашата програма наистина не може да направи нищо друго.

Полезен съвет: Планирайте как ще обработвате изключенията!

Когато проектирате Вашата програма трябва да обмисляте кои събития се броят като стопери и как можете да се справите с тях най-елегантно.

Конструкцията switch

C# съдържа специална програмна конструкция, позволяваща ни да изберем опция от няколко възможни стойности, на базата на точно определена една от тях. Това е конструкцията **-switch**:

```
switch (selection)
{
    case 1:
        Execution1();
        break;
    case 2:
        Execution2();
        break;
    case 3 :
        Execution3();
        break;
    default: Console.WriteLine("Грешен избор!");
    break;
}
```

Конструкцията **switch** взема стойност, която използва за да реши коя опция да извърши. Изпълнява се случая, който съвпада със стойността на променливата.

Програмната конструкция **break** след извикването на съответния метод е необходима, за да се преустанови по-нататъшно изпълнение на конструкцията **switch** и вместо това да се премине на кода, който следва след нея. По същия начин, по който спираме цикъл, когато се стигне до **break**, оператор **switch** приключва и програмата продължава с изпълнението на програмните конструкции непосредствено след него.

Друга полезна особеност е опцията по подразбиране **default**. Това дава на оператора **switch** място където да отиде, ако стойността не съвпада с никой от наличните преди това случаи.

Можете да използвате програмната конструкция **switch** с други типове освен числа, например:

```

switch (selection)
{
    case "едно":
        Execution1();
        break;
    case "две":
        Execution2();
        break;
    case "три":
        Execution3();
        break;
    default:
        Console.WriteLine("Грешна команда!");
        break;
}

```

Този пример използва низ, за да контролира избора на случаи.

Полезен съвет: Switch е добра идея!

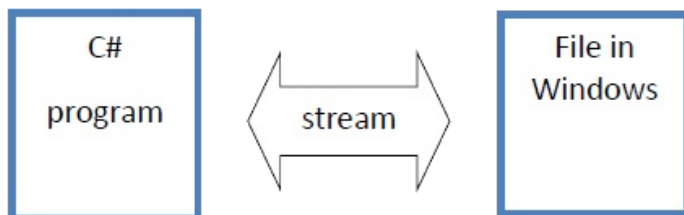
Тази конструкция може да направи програма по-лесна за разбиране, както и по-бърза за написване. Също така е по-лесно да се добавят допълнителни команди, ако използвате switch, понеже става въпрос просто за слагане на още един допълнителен случай. Въпреки това бих Ви посъветвал да не слагате голямо количество програмен код в case блок. Вместо това би трябвало да поставите извикване на метод, както е показано в примерите по-горе.

Използване на файлове

C# използва **потоци**³², за да позволи на програмите да работят с файлове. Поток е връзката между Вашата програма и данни на ресурсите. Данните могат да текат нагоре или надолу по Вашия поток, така че потоците могат да се използват за четене и писане от и във файлове.

³²Потоците са свързани входно изходни канали между компютърната програма и неговата среда. Термин на Английски език = **stream**, вж.

http://en.wikipedia.org/wiki/Standard_streams



Програма на C# може да съдържа обект представляващ определен поток, който програмиста е създал и свързал към файл. Програмата изпълнява операции върху файла като извиква методи на поток обекта, които й казват какво да прави.

Създаване на изходен поток

Създаваме обект от тип поток точно както бихме създали всеки друг обект – посредством оператора **new**. Когато създаваме поток, можем да му подадем името на файла, който трябва да бъде отворен.

```
StreamWriter writer = new StreamWriter("test.txt");
```

Създава се променлива **writer**, която указва потока, в който искаме да пишем. Когато се създава новият **StreamWriter**, програмата ще отвори файл наречен **test.txt** и ще свърже потока към него. Ако този процес се провали, поради каквато и да е причина, тогава действието ще бъде преустановено с подходящото изключение.

Писане в поток

Веднъж след като потока е създаден в него може да се пише чрез извикване на методите за писане, които той осигурява:

```
writer.WriteLine("Здравей свят!");
```

Това е абсолютно същата техника, която се използва при извеждане на информация в конзолата, която потребителя чете. Всеки път когато записвате ред във файла, то той се добавя в

края, след последните записани редове.

Затваряне на поток

Много е важно, когато Вашата програма свърши да пише в потока, той да бъде стриктно затворен използвайки метод **close**.

```
writer.Close();
```

След това, всеки друг опит да пишем в потока ще се провали с изключение за грешка.

Забравянето на затваряне на файл е лоша практика поради редица причини:

- Възможно е програмата да приключи своята работа без файла да е затворен правилно. В тази ситуация част от данните, които сте записали във файла може да не са там.
- Ако програмата има отворен поток, свързан към файл, други програми няма да са способни да използват този файл. Също така ще е невъзможно да преместите или преименувате файла.
- Отвореният поток изразходва малка, но значима част от операционните ресурси. Ако Вашата програма създава много потоци, но не ги затваря, това може да доведе до проблеми при отварянето на други файлове по-късно.

Потоци и пространства от имена

Обектите, които работят с входа и изхода са дефинирани в пространството от имена *System.IO*. С други думи, за да използвате класовете, които работят с файлове, трябва да добавите следното в началото на Вашата програма:

```
using System.IO;
```

Четене от файл

Четенето от файл е много подобно на писането, по начина по който програмата ще създаде поток извършващ същинската

работа. В този случай обаче потока, който се използва е **StreamReader**.

```
TextReader reader = new StreamReader("Test.txt");
string line = reader.ReadLine();
Console.WriteLine (line);
reader.Close();
```

Горната програма свързва поток към файла **Test.txt**, чете първият ред от файла, показва го на екрана и след това затваря потока. Ако файлът не може да бъде намерен, тогава опита той да бъде отворен ще се провали и програмата ще хвърли изключение за грешка.

Откриване на края на входен файл

Повторни извиквания на **ReadLine** ще връщат последователни редове от файла. Ако обаче програмата Ви стигне края на файла, методът **ReadLine** ще връща празен низ всеки път, когато той бъде извикан повторно. За щастие обекта **StreamReader** осигурява свойство наречено **EndOfStream**, което програмата може да използва, за да разбере кога е достигнат края на файла. Когато свойството стане вярно, то края на файла е достигнат.

```
StreamReader reader;
reader = new StreamReader("Test.txt");
while (reader.EndOfStream == false)
{
    string line = reader.ReadLine();
    Console.WriteLine(line);
}
reader.Close();
```

Горната програма ще отвори файла **Test.txt** и ще изведе всеки ред от файла на конзолата. Цикълът **while** ще спре програмата, когато е достигнат края на файла.

Път до файла в C#

Ако искате да използвате файл в различна папка, то можете да добавите информация за пътя към името на файл по следния

начин:

```
string path = path = @"c:\data\2009\November\sales.txt";
```

Горният израз създава променлива от тип **string**, която съдържа пътя към файл наречен **sales.txt**. Този файл се съдържа в папката **November**, която на свой ред се намира в папката **2009**, която от своя страна е в папката **data**, записана на диск **C**.

Символът наклонена черта (\) в низа служи, за да раздели папките по пътя към файла. Забележете, че съм уточнил низа да е буквален, т.е. да не съдържа контролни символи (*това означава литералът за буквален израз @ поставен в началото на низа*), защото в противен случай символа \ в низа, ще бъде интерпретиран от C#, като начало на контролна поредица.

ГЛАВА 4. СЪЗДАВАНЕ НА РЕШЕНИЯ

Настоящата глава задълбочава познанията чрез: създаване и използване на структури; работа с класове и обекти, техните инстанции и референции; използване на компоненти, интерфейс и дизайн; наследяване; потискане на методи; виртуални методи; конструктори и йерархии; абстрактни методи и класове; референции към абстрактни класове; проектиране на обекти и компоненти; свойства, интерфейси и делегати.

Създава се цяло банково приложение използвайки езика за програмиране C# и се разглеждат особеностите му, които помагат да се разработва лесно.

Обхват на банковата система

Обхват³³ на една система е описание на какво системата може да извършва. Също така по импликация е в сила обратното твърдение, какво системата няма да може да прави. Това е еднакво важно, понеже клиента обикновено няма ясна представа какво Вие правите и може да очаква да му доставите функционалност, която нямате намерение. Чрез определяне обхвата на системата в началото може да сте уверени, че няма да има неприятни изненади на по-късен етап от работата.

Управителят на банката ни е казал, че банката пази информация за всеки свой клиент. Тази информация включва: името на клиента, неговият адрес, номер на сметка, текущ баланс и стойност на кредита. Други данни могат да бъдат добавяни на по-късен етап. Има хиляди клиенти и управителят също ни е казал, че има и различни видове сметки и освен това нови

³³Термин на Английски език = **scope**, *вж.*

[http://en.wikipedia.org/wiki/Scope_\(computer_science\)](http://en.wikipedia.org/wiki/Scope_(computer_science))

видове сметки биват измислени от време на време. Системата също така трябва да генерира предупредителни писма и отчети, ако е необходимо.

Изброени типове

C# има начин, по който можем да създадем тип данни съдържащ просто определено множество от възможни стойности. Тези типове се наричат **изброени типове**³⁴:

```
enum Light
{
    Red,
    Yellow,
    Green
};
```

Създадох тип наречен *Light*, който може да бъде използван да съдържа състоянието на светофар. Той може да има само дадените по-горе стойности и трябва да бъде управляван единствено по отношение на тези именувани номерации. Например:

```
Lighttest;
test = Light.Green;
```

Важно е да разберете какво се случва тук. Преди това използвахме типове, които са част от C#, например *int* и *double*. Сега достигнахме момента, където ние действително създаваме наши собствени типове за данни, които могат да бъдат използвани за да съдържат стойности, които се изискват от нашето приложение.

Създаване на тип enum

Новият тип **enum** може да бъде създаден извън всеки клас и

³⁴Изброени типове, съдържат краен брой опции. Термин на Английски език = **Enumerated Types**, вж. http://en.wikipedia.org/wiki/Enumerated_types и <http://msdn.microsoft.com/en-US/library/vstudio/cc138362.aspx>

създава нов тип, който може да се използва във всяка програма:

```
using System;

enum Light
{
    Red,
    Yellow,
    Green
};

class EnumDemonstration
{
    public static void Main ()
    {
        Light test;
        test = Light.Red;
    }
}
```

За банката искаме да знаем състоянието, както и друга информация за клиента. Например можем да имаме състоянията: „нов“, „активен“, „в проверка“, „замразен“ и „затворен“, като различни възможни състояния за банковата сметка на клиента.

```
enum AccountState
{
    New,
    Active,
    UnderAudit,
    Frozen,
    Closed
};
```

Сега имаме променлива, която съдържа информация за състоянието на сметка в банката.

Проста структура

От спецификацията вече знаем, че програмата трябва да съдържа следното:

- име на клиента – текстов низ;
- адрес на клиента – текстов низ;

- номер на сметката – цяло число;
- баланс на сметката – цяло число;
- лимит на кредита – цяло число.

Банката е казала, че те ще слагат само до 50 човека във Вашата банкова памет, така че след известно време стигаме до следния програмен код:

```
const int MAX_CUST = 50;
AccountState [] states = new AccountState [MAX_CUST];
string [] names = new string [MAX_CUST];
string [] addresses = new string [MAX_CUST];
int [] accountNos = new int [MAX_CUST];
int [] balances = new int [MAX_CUST];
int [] overdraft = new int [MAX_CUST];
```

Всичко това е много хубаво и можете да накарате системата за база от данни да работи с тази структура от данни. Много по-добре обаче ще бъде, да съберете целия този запис заедно по друг по-точно дефиниран начин.

Създаване на структура

C# ни позволява да създаваме структура от данни. Структурата е колекция от променливи, които искаме да разглеждаме като една същност.

```
struct Account
{
    public AccountState State;
    public string Name;
    public string Address;
    public int AccountNumber;
    public int Balance;
    public int Overdraft;
};
```

Този код дефинира структура, наречена **Account**, която съдържа цялата необходима информация за клиента. След като сме направили това, вече можем да дефинираме някои променливи:

```
Account MinchevAccount;
Account [] Bank = new Account[MAX_CUST];
```

Първата декларация създава променлива наречена **MinchevAccount**, която съдържа информация за един клиент. Втората декларация създава цял масив от клиенти наречен **Bank**, който може да съдържа информация за всички клиенти на банката.

Използване на структура

```
using System;
enum AccountState
{
    New,
    Active,
    UnderAudit,
    Frozen,
    Closed
};
struct Account
{
    public AccountState State;
    public string Name;
    public string Address;
    public int AccountNumber;
    public int Balance;
    public int Overdraft;
};
class BankProgram
{
    public static void Main()
    {
        Account MinchevAccount;
        MinchevAccount.State = AccountState.Active;
        MinchevAccount.Name = "Minchev";
        MinchevAccount.Address = "Burgas";
        MinchevAccount.AccountNumber = 1234;
        MinchevAccount.Balance = 0;
        MinchevAccount.Overdraft = -1000;
        Console.WriteLine ("Name is: "+MinchevAccount.Name);
        Console.WriteLine ("Balance is:
"+MinchevAccount.Balance);
    }
}
```

Използване на тип структура при извикване на метод

```
public void PrintAccount (Account acc)
{
    Console.WriteLine ("Name: "+acc.Name);
    Console.WriteLine ("Address: "+acc.Address);
    Console.WriteLine ("Balance: "+acc.Balance);
}
```

Този метод ни позволява бърз начин да отпечатваме съдържанието на променливата **account**:

```
PrintAccount(MinchevAccount);
```

Полезен съвет: Структурите са от решаващо значение

В комерсиална система, често се отделя много дълго време за проектиране на структурите, които ще съхраняват данните. Те са основните градивни елементи на програмата, понеже съдържат данните, на които се крепи всичко останало. Може да разглеждате проектирането на структури и ограниченията за тяхното съдържание, като друга голяма буца от мета данни относно системата, която създавате.

```
enum windowState
{
    Quoted,
    Ordered,
    Manufactured,
    Shipped,
    Installed
};

struct window
{
    public windowState state;
    public double width;
    public double height;
    public string Description;
};
```

Това би трябвало да съдържа информация относно определен прозорец в една къща, включвайки размерите на прозореца, състоянието на поръчката за прозореца и низ с описание на

самия прозорец. За определена къща, която съдържа няколко прозореца бихме създали масив от структури от типа **Window**.

Обекти и структури

Структурите се управляват по отношение на стойности, за разлика от обектите, които се управляват по отношение на референции.

```
struct AccountStruct
{
    public string Name;
};

class StructsAndObjectsDemo
{
    public static void Main()
    {
        AccountStruct MinchevAccountStruct;
        MinchevAccountStruct.Name = "Minchev";
        Console.WriteLine(MinchevAccountStruct.Name);
    }
}
```

Този програмен код реализира много проста банкова сметка, в която съхраняваме името на собственика на сметката.

Създаване и използване на инстанция на клас

```
class Account
{
    public string Name;
};

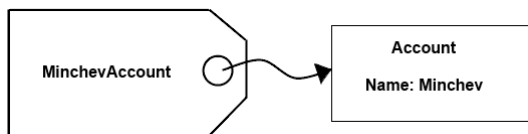
class StructsAndObjectsDemo
{
    public static void Main()
    {
        Account MinchevAccount;
        MinchevAccount.Name = "Minchev";
        Console.WriteLine(MinchevAccount.Name);
    }
}
```

Информацията за акаунта сега се съдържа в клас, вместо в структура. Проблемът е, че когато компилираме програмата получаваме:

```
ObjectDemo.cs(12,3): error CS0165: Use of unassigned local variable 'MinchevAccount'
```

Това, което всъщност се случва, когато програмата изпълни този ред е създаване на **референция**³⁵, наречена **MinchevAccount**. Създавайки референцията, всъщност не взимаме едно от нещата, към които тя се отнася. Компиляторът следователно казва: *„Вие се опитвате да проследите референция, която не се отнася към нищо, следователно аз ще ви върна грешка неидентифицирана променлива“*. Този проблем може да бъде решен като създаваме инстанция на класа и тогава свързваме нашия етикет с него. Това се осъществява като добавим следния ред в програмата:

```
Account MinchevAccount;  
MinchevAccount = new Account();
```

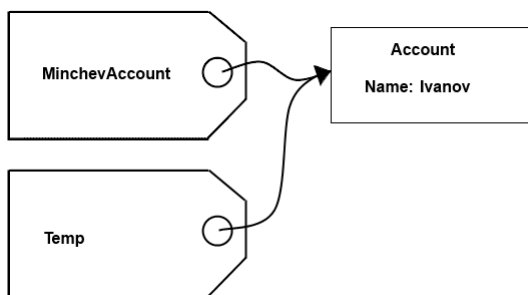


Виждали сме тази ключова дума и преди. Използваме я, за да създаваме масиви. Това е така, защото масива всъщност е реализиран като обект и за това използваме оператор **new**, за да го създадем. **New** създава обект. Обектът е отделен случай на класа.

³⁵Референцията, всъщност е указател към обект. Тя посочва или по-точно сочи към обекта. Може също да се каже, че референцията е връзка към инстанция на клас. Термин на Английски език = **reference**

Множество референции към инстанция

```
Account MinchevAccount;  
MinchevAccount = new Account();  
MinchevAccount.Name = "Minchev";  
Console.WriteLine(MinchevAccount.Name);  
Account Temp;  
Temp = MinchevAccount;  
Temp.Name = "Ivanov";  
Console.WriteLine(MinchevAccount.Name);
```



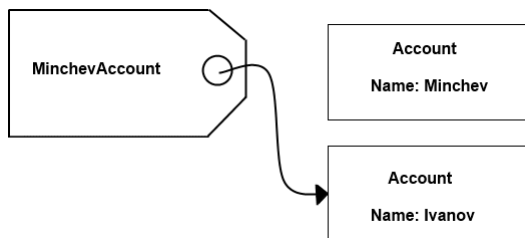
И двата етикета реферират една и съща инстанция на **Account**. Това означава, че всякакви промени, които бъдат направени към обекта, който **Temp** реферира, ще бъдат отразени и в този, който **MinchevAccount** реферира, защото те практически са един и същи обект.

Без референции към инстанция

За да запълним объркването още повече, трябва да разгледаме, какво се случва, ако обект няма референция към него:

```
Account MinchevAccount;  
MinchevAccount = new Account();  
MinchevAccount.Name = "Minchev";  
Console.WriteLine(MinchevAccount.Name);  
MinchevAccount = new Account();  
MinchevAccount.Name = "Ivanov";  
Console.WriteLine(MinchevAccount.Name);
```

Въпросът, който възниква тук е какво се случва с първата инстанция?



И отговорът е тривиален: „Нищо!“. Първата инстанция е показана рееща се в космоса, защото няма референция към нея. В езикът С# има специален процес наречен **боклуджията**³⁶, чиято задача е да намира такива безполезни неща и да се отървава от тях.

Данни в обектите

Първото нещо, което трябва да направим е да идентифицираме всички необходими данни, които искаме да съхраняваме в даден клас.

```
class Account
{
    public decimal balance;
}
```

Класът **Account** по – горе съдържа член данни, който ни е необходим, за да запазваме баланса на нашите банкови сметки. Член данни на клас, които съдържат стойност, описваща някакви данни, които класът съдържа, често се наричат **свойства**³⁷.

Видяхме, че всеки един от елементите съдържащи се в даден клас, всъщност представлява член данни на този клас, и също така тези елементи се съхраняват като част от класа. Освен това

³⁶Боклуджията, всъщност е процес още познат като „събиране на боклука“.

Термин на Английски език = **Garbage Collector**, вж.

<http://msdn.microsoft.com/en-us/library/Oxy59wtx.aspx>

³⁷Термин на Английски език = **properties**

всеки път, когато създаваме инстанция на класа, то поучаваме в комплекта и всички участници в този клас. Вече видяхме, че е много лесно да се създаде инстанция на клас и да се зададе стойност в член данни от класа:

```
Account MinchevAccount;  
MinchevAccount = new Account();  
MinchevAccount.Balance = 99;
```

Причината поради която работи е, че членовете на обекта са **публични**³⁸, което означава, че всички имат директен достъп до тях.

Защита на данните в обектите

Ако искаме обектите да бъдат полезни, то трябва да имаме начин да защитим данните съдържащи се в тях. Елегантната дума, която се използва за това е **капсулиране**³⁹. Тази технология е ключът към моя защитен програмен подход, които е насочен да гарантира, че каквото и да стане моята част от програмата няма да се обърка.

Първото нещо, което трябва да направим е да спрем останалия свят да си играе със стойността, която се съхранява в променливата *balance*. Това свойство вече не е маркирано като публично. Вместо това сега то е **частно**⁴⁰. Това означава, че

³⁸Публични член данни на класа, са общодостъпните елементи участващи в този клас. Общодостъпни ще рече, че всеки има директен достъп до тях и тяхното съдържание. Термин на Английски език = **public**, вж.

<http://msdn.microsoft.com/en-us/library/yzh058ae.aspx>

³⁹Капсулиране е процес на сиване на определена информация само за употреба вътре в класа. Термин на Английски език = **encapsulation**, вж. [http://en.wikipedia.org/wiki/Encapsulation_\(object-oriented_programming\)](http://en.wikipedia.org/wiki/Encapsulation_(object-oriented_programming)) и <http://www.codeproject.com/Tips/458785/Pillars-of-OOPS-Part-1-Encapsulation>

⁴⁰Частно свойство или частни член данни, са тези елементи на класа, които ще се консумират само от него. Никой друг няма да има директен достъп до тях и тяхното съдържание освен други член данни на същия този клас. Термин от

Останалия свят вече няма директен достъп до него.

```
class Account
{
    private decimal balance = 0;
    public bool WithdrawFunds(decimal amount)
    {
        if(balance < amount)
        {
            return false;
        }
        balance = balance-amount;
        return true;
    }
};

class Bank
{
    public static void Main()
    {
        Account MinchevAccount;
        MinchevAccount = new Account();
        if(MinchevAccount.WithdrawFunds(5))
        {
            Console.WriteLine("Cash withdrawn");
        }
        else
        {
            Console.WriteLine("Insufficient Funds");
        }
    }
}
```

Този програмен фрагмент създава банкова сметка и се опитва да изтегли пет евро от нея. Това действие разбира се, ще се провали, понеже началния баланс на сметката е нула, но показва как се осъществява достъп до член данните на даден клас. Методът *WithdrawFunds* е член на класа *Account* и заради това има достъп до частните член данни в него.

Публични методи

Може би сте забелязали, че направих метод *WithdrawFunds*

Английски език = **private**, вж. <http://msdn.microsoft.com/en-us/library/st6sy9xe.aspx>

публичен. Това означава, че кода, който е извън класът може да извиква този метод.

Общо взето правилата при работа с класове са следните:

- ако става въпрос за **член данна**⁴¹ на класа, то направете я частна;
- ако става въпрос за **член функция**⁴² на класа, то направете я публична.

Съвет: Използвайте кодови конвенции, за да покажете кои неща са частни

Ако пиша името на публичен елемент, започвам с главна буква в началото на името. Но първата буква на частен елемент пиша с малка буква. Това ще улесни този, който чете моя код, защото лесно по името на член данните ще разбере дали даден елемент е публичен или частен. Всъщност повечето компании за разработка имат документи, които посочват кодовите конвенции, които се използват и очакват разработчиците да се придържат към тях.

⁴¹Член данна = елемент на класа съдържащ данни.

⁴²Член функция = метод в класа, който прави нещо.

Завършен клас за сметки

```
public class Account
{
    private decimal balance = 0;
    public bool withdrawFunds(decimal amount)
    {
        if(balance < amount)
        {
            return false;
        }
        balance = balance-amount;
        return true;
    }

    public void PayInFunds(decimal amount)
    {
        balance = balance+amount;
    }

    public decimal GetBalance()
    {
        return balance;
    }
}
```

Мога да напиша малко код, за да пробвам методите на класа:

```
Account test = new Account();
test.PayInFunds(50);
if(test.GetBalance() != 50)
{
    Console.WriteLine("Pay In test failed");
}
```

Програмата сега се само проверява, т.е. тя прави нещо, след което се уверява, че ефекта на това действие е правилен.

Разработка базирана на тестове⁴³

1. Не правете тестването в края на проекта. Обикновено това е най-лошото време за тестване, понеже може да използвате код, който сте написали преди известно време. Ако бъговете се намират в стара част от програмния код, то трябва да

⁴³Разработка базирана на тестове, фраза на Английски език = **Test Driven Development**, вж. http://en.wikipedia.org/wiki/Test_Driven_Development

преминете през усилията да си припомните как той действа. Много по-добре е да тествате кода веднага след като го напишете, когато имате възможно най-доброто разбиране на това какво този програмен код трябва да прави.

2. Може да напишете код, в по-ранен етап на проекта, който вероятно ще бъде полезен по-късно. Много проекти са обречени, понеже хората започват да програмират преди да имат ясна представа за проблема. Писането първо на тестове, всъщност е наистина добър начин да усъвършенствате Вашето разбиране за проекта. И също така има добър шанс тестовите, които направите да бъдат полезни на даден етап.
3. Когато оправяте бъгове във Вашата програма ще трябва да се убедите, че поправките не са развалили някоя друга част от кода (*един от най-често срещаните начини да се появят нови неизправности в програмата е да поправите бъг*). Ако имате автоматични тестове, които работят след всяка поправка на бъгове, то имате начин да спрете това.

Така че, моля започнете да разработвате като използвате тестове.

Ще ми благодарите по-късно.

Статични членове на клас

Ключовата дума *static* ни позволява да създаваме членове, които се съдържат не в инстанция, а в самия клас. Ето кратък програмен фрагмент за пояснение:

```

class AccountTest
{
    public static void Main()
    {
        Account test = new Account();
        test.PayInFunds(50);
        Console.WriteLine("Balance:"+test.GetBalance());
    }
}

```

Ако направя петдесет инстанции на класа *AccountTest*, всички те ще споделят общия за тях метод *Main*. Ключовата дума *static* в С# показва, че метода е част от класа, а не част от инстанцията му. Всъщност всичко това означава, че не е нужно да правя инстанция на класа *AccountTest*, за да мога да използвам метода *Main*. Методът е винаги там и така моята програма всъщност работи.

Static не означава „не може да бъде променен“.

Използване на статични член данни в клас
 Казаха ми, че лихвеният процент е валиден за всички банкови сметки. Ако той се промени, то това трябва да бъде отразено в тях. Това означава, че за да реализирам тази промяна трябва да премина през всички сметки и да обнова стойността им. Вместо това решавам този проблем като правя члена съдържащ стойността на лихвения процент статичен, по следния начин:

```

public class Account
{
    public decimal balance;
    public static decimal InterestRateCharged;
}

```

Лихвеният процент вече е част от самият клас, а не от която и да е инстанция. Това означава, че трябва да променя също така и начина по който я използвам:

```
Account MinchevAccount = new Account();
MinchevAccount.Balance = 100;
Account.InterestRateCharged = 10;
```

Понеже лихвеният процент е вече член на класа, сега трябва да използвам името на класа, вместо референцията към инстанцията.

Използване на статичен метод в клас

```
public static bool AccountAllowed(decimal income,int age)
{
    if((income >= 10000)&&(age>=18))
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

В по горния пример метода е част от класа, а не част от инстанцията му. Сега мога да извикам метода, като използвам името на класа.

Използване на член данни в статични методи

```
public class Account
{
    private decimal minIncome = 10000;
    private int minAge = 18;
    public static bool AccountAllowed(decimal income,int age)
    {
        if((income >= minIncome)&&(age >= minAge))
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

Това обаче е лоша програма, понеже горния клас няма да се компилира.

Както обикновено, компилатора ни казва точно какво се е объркало, използвайки език, който кара главите ни да се завъртят. Това, което компилаторът наистина има в предвид е, че *„статичен метод използва член от класа, който не е статичен“*.

Обаче статичен метод може да работи без инстанция (*понеже е част от класа*). Компилаторът не е щастлив, защото в тази ситуация метода няма да има никакви членове, с които да си играе. Може да поправим това (*и да накараме програмата да бъде абсолютно вярна*) като направим *income* и *age* също статични:

```
public class Account
{
    private static decimal minIncome;
    private static int minAge;
    public static bool AccountAllowed(decimal income,int age)
    {
        if((income >= minIncome)&&(age >= minAge))
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

Ако се замислите тук има перфектен смисъл.

Статичните член данни могат да бъдат използвани за създаване на библиотеки

Понякога по време на разработката ще се наложи да осигурите библиотека от методи, които да правят разни неща. В C# има огромен брой методи, които да изпълняват различни функции, например математически, като *sin* и *cos*. Има смисъл тези методи да са статични, защото в тази ситуация всичко, което искаме е самия метод, а не инстанция на клас. Когато изграждате Вашата програма трябва да мислите как да направите така, че методите Ви да се налични за собствено използване.

Създаване на обекти

Когато се създава инстанция на клас, C# извиква **конструктор**⁴⁴ метода на този клас. Конструктор метода е член от класа и е там, за да позволи на програмиста да поеме контрол и да създаде съдържанието на новосъздадения обект.

Конструктор по подразбиране

Конструкторът има същото име както класа, но не връща нищо. Извиква се, когато изпълняваме оператора *new*. Ако не предложите⁴⁵ конструктор (*а ние до сега не сме*), компилаторът създава автоматично такъв за нас.

```
public class Account
{
    public Account()
    {
    }
}
```

Така изглежда конструктор по подразбиране. Той е публичен,

⁴⁴Конструктор е специален метод носещ името на класа. Програмният код в блока на този метод се изпълнява, когато се създаде обект екземпляр на този клас. Термин на Английски език = **Constructor**, вж.

[http://en.wikipedia.org/wiki/Constructor_\(object-oriented_programming\)](http://en.wikipedia.org/wiki/Constructor_(object-oriented_programming))

⁴⁵Тук множество значения имат смисъл. Можете да: предложите, доставите, създадете или произведете конструктор. Превод от Английски език = **Supply**

така че да бъде достъпен от външни класове, които трябва да могат да направят инстанция на класа.

```
minchevAccount = new Account();
```

Предефиниран конструктор

Предефиниран⁴⁶ е интересна дума. В контекста на C# програмата означава: *„Методът има същото име като друг метод, но има различен комплект параметри“*.

```
minchevAccount = new Account("Dimitar Minchev", "Burgas Free University");  
public Account(string inName, string inAddress)  
{  
    name = inName;  
    address = inAddress;  
    balance = 0;  
}
```

Предефиниране на методи

Всъщност можете да предефинирате всяко име на метод във Вашите класове:

```
SetDate(int year,int month,int day)  
SetDate(int year,int julianDate)
```

Извикването на предефиниран метод може да се осъществи така:

```
SetDate(2005,7,23);
```

Конструктора не може да се провали

Ако сте гледали филм с Джеймс Бонд, обикновено има момент, в който се казва на агента, че съдбата на света е в неговите ръце и неуспехът не е опция. Конструкторите са нещо такова.

⁴⁶Предефиниран конструктор, съществува втори метод като Конструктора, даже носещ неговото име, но използващ различен брой параметри. Терминът може да се преведе и като: претоварен конструктор. Термин на Английски език = **Overloaded constructor**

Конструкторите не могат да се провалят. И това е проблем!

От обект към компонент

Важен момент, който трябва да отбележим тук е, че не говорим за потребителски интерфейс на програмата, който обикновено е текстов или графичен. Интерфейсът в конкретния контекст, определя как един софтуерен компонент може да бъде използван от друг софтуерен компонент.

Интерфейс и дизайн

Така, че вместо да започнем да проектираме класове, трябва да мислим за описване на техния **интерфейс**⁴⁷, тоест какво е онова нещо, което те ще правят. В C# изразяваме тази информация в така наречения интерфейс. Интерфейсът е просто група дефиниции на методи, които са залепени заедно. Ето примерен интерфейс:

```
public interface IAccount
{
    void PayInFunds(decimal amount);
    bool WithdrawFunds(decimal amount);
    decimal GetBalance();
}
```

Имплементиране на интерфейс

Интерфейсите стават интересни, когато направим клас, който да ги имплементира. Ако клас имплементира интерфейс то се казва, че за всеки метод описан в интерфейса на класа, има съответна имплементация.

⁴⁷Интерфейсът описва група от свързани поведения, които могат да принадлежат към всеки клас или структура. Интерфейсите могат да се състоят от методи, свойства, събития, идентификатори или всяка тяхна комбинация. Термин на английски език = **Interfaces**, вж. [http://msdn.microsoft.com/en-US/library/ms173156\(v=vs.80\).aspx](http://msdn.microsoft.com/en-US/library/ms173156(v=vs.80).aspx)

```

public class CustomerAccount : IAccount
{
    private decimal balance = 0;

    public bool withdrawFunds(decimal amount)
    {
        if(balance < amount)
        {
            return false;
        }
        balance = balance-amount;
        return true;
    }

    public void PayInFunds(decimal amount)
    {
        balance = balance+amount;
    }

    public decimal GetBalance()
    {
        return balance;
    }
}

```

Референции към интерфейси

Веднъж след като сме компилирали класа *CustomerAccount*, вече имаме нещо, което може да се разглеждаме по два начина:

- като *CustomerAccount*, защото това е което е: клиентска сметка;
- като *Iaccount*, защото това е което прави: интерфейс на сметката.

Във фразеологията на C# това означава, че трябва да умеем да създаваме променливи референции, които да сочат към обекти, в смисъл на интерфейса, който те имплементират, вместо към определения тип, който всъщност са. Оказва се, че това е доста лесно:

```

IAccount account = new CustomerAccount();
account.PayInFunds(50);

```


Използване на интерфейси

Сега, след като нашата система е проектирана с интерфейси, вече е много по-лесно да я разширим. Можем да създадем нов клас *BabyAccount*, който да имплементира интерфейса на *Iaccount*.

```
public class BabyAccount : IAccount
{
    private decimal balance = 0;

    public bool withdrawFunds(decimal amount)
    {
        if(amount > 10)
        {
            return false ;
        }
        if(balance < amount)
        {
            return false ;
        }
        balance = balance-amount;
        return true;
    }

    public void PayInFunds(decimal amount)
    {
        balance = balance+amount;
    }

    public decimal GetBalance()
    {
        return balance;
    }
}
```

Хубавото в това е, че понеже е компонент не трябва да променяме всички класове, които той използва. Когато създаваме обекти от тип *account* просто трябва да попитаме дали се изисква стандартен *account* или *babyaccount*.

Имплементиране на множество интерфейси

Един компонент може да имплементира толкова интерфейса от колкото има нужда:

```
public interface IPrintToPaper
{
    void DoPrint();
}
```

Сега всичко, което имплементира интерфейса *IPrintToPaper* ще съдържа метод *DoPrint* и може да се мисли от гледна точка на способността му да отпечатва.

Един клас може да имплементира толкова интерфейси от колкото има нужда.

```
public class BabyAccount : IAccount, IPrintToPaper { ...
```

Интерфейсите са просто обещания

Интерфейсите не са обвързващ договор, а са по-скоро обещание. Просто защото един клас има метод наречен PayInFunds не означава, че той ще превежда пари в сметка; просто означава, че метод с това име съществува в този клас.

Механизмът на интерфейсите ни дава голяма степен на гъвкавост, когато правим нашите компоненти и ги сглобяваме заедно. Това означава, че след като сме открили това, което нашия банков клас трябва да съдържа, можем да се замислим какво да накараме сметките да правят. Това всъщност е действителната подробност на спецификацията. Когато сме настроили интерфейс за компонент можем просто да мислим от гледна точка, какво компонента трябва да прави, а не точно как да го направи.

Наследяване

На кратко:

- Интерфейс: „Мога да правя тези неща, защото съм казал,

че мога.“;

- **Наследяване**⁴⁸: „Мога да правя тези неща, защото моя родител може.“.

Разширяване на родителски клас

```
public class BabyAccount : CustomerAccount, IAccount
{
}
```

Ключово тук е частта след името на класа. Сложили сме името на класа, който разширява *BabyAccount*. Това означава, че всичко, което *CustomerAccount* може да прави, може да бъде направено и от *BabyAccount*.

```
BabyAccount b = new BabyAccount();
b.PayInFunds(50);
```

Припокриване на методи

Следващото, което искаме е да промениме поведението на един от методите, който ни интересува. Това се нарича **припокриване на метод**⁴⁹. Както е показано в примера:

```
public class BabyAccount : CustomerAccount, IAccount
{
    public override bool withdrawFunds(decimal amount)
    {
        if (amount > 10) return false;
        if (balance < amount) return false;
        balance = balance - amount;
        return true;
    }
}
```

Ключовата дума **override** означава: „използвай тази версия на

⁴⁸Наследяването е част от обектно - ориентираното програмиране. Термин на Английски език = **Inheritance**, вж. [http://msdn.microsoft.com/en-us/library/ms173149\(v=vs.80\).aspx](http://msdn.microsoft.com/en-us/library/ms173149(v=vs.80).aspx)

⁴⁹Припокриване на методи, означава промяна на първоначалната им функционалност, т.е. пренаписване на блока от код, който тези методи изпълняват. Термин на Английски език = **Overriding methods**, вж. <http://www.codeproject.com/Articles/18734/Method-Overriding-in-C>

метод за предпочитане вместо тази в родителя“.

```
BabyAccount b = new BabyAccount();  
b.PayInFunds(50);  
b.WithdrawFunds(5);
```

Извикването на *PayInFunds* ще използва метода в родителя, но извикването на *WithdrawFunds* ще използва метода в *BabyAccount*.

Виртуални методи

Компилаторът на C# трябва да знае, че определен метод ще бъде припокрит. За да работи припокриването на метод правилно, трябва да сменим също така и декларацията на *WithdrawFunds* в класа *CustomerAccount*, като го направим **виртуален метод**⁵⁰. Както е показано в примера:

```
public class CustomerAccount : IAccount  
{  
    private decimal balance = 0;  
    public virtual bool withdrawFunds(decimal amount)  
    {  
        if(balance < amount)  
        {  
            return false;  
        }  
        balance = balance-amount;  
        return true;  
    }  
}
```

Ключовата дума **virtual** означава буквално: „Може да искам да направя друга версия на този метод в класа наследник“.

*Това прави **override** и **virtual** нещо като съвместно работеща двойка. Използваме **virtual** за да маркираме метод като способен да бъде припокрит и **override** действително, за да предоставим заместник за този метод.*

⁵⁰Виртуалният метод маркира метода, като способен да бъде припокрит.

Термин на Английски език = **Virtual methods**, вж.

<http://www.codeproject.com/Articles/16226/Virtual-Extension-Methods>

Защита на данните в йерархия от класове

Всъщност горният програмен код отново няма да работи. Това е така, защото стойността на променливата за баланс в класа е обявена със спецификатор за достъп частни член данни. Обаче тази защита е прекалено стриктна. За да заобиколим този проблем C# ни предоставя значително по-малко ограничително ниво на достъп наречено **защитени**⁵¹ член данни. Това прави член данните видими за класа наследник на родителския клас. С други думи клас може да вижда и използва защитени членове, защото те са в същата йерархия на класа, който съдържа тези член данни.

```
public class CustomerAccount : IAccount
{
    protected decimal balance = 0;
    .....
}
```

Използване на базов метод

Дизайнерите на C# са осигурили начин, по който може да извикате базов метод от някой, който го припокрива. Думата базов в този контекст означава „*препратка към нещо, което е било припокррито*“.

⁵¹Спецификатор за достъп: защитени член данни. Позволява употребата на тези член данни при йерархично наследяване на класове, като същевременно ги защитава от външен достъп. Термин на Английски език = **protected**, вж. <http://msdn.microsoft.com/en-us/library/bcd5672a.aspx>

```

public class BabyAccount : CustomerAccount, IAccount
{
    public override bool withdrawFunds(decimal amount)
    {
        if(amount > 10)
        {
            return false ;
        }
        return base.withdrawFunds(amount);
    }
}

```

По този начин са решени следните проблеми:

- Не искам да пиша същия код два пъти.
- Не искам да правя стойността на баланса да е видима извън класа *CustomerAccount*.

Използването на думата **base**, за да извикаме припокрит метод решава и двата проблема доста красиво. Понеже извикването на метода връща булев резултат, то може просто да се изпрати това, което той доставя. Правейки тази промяна, може отново да върнем баланса на *private* в *CustomerAccount*, защото променливата не се изменя извън него.

Конструктори и йерархии

Конструкторът е метод, който поема контрол по време на процеса на създаване на обект от този клас. Използва се от програмиста за задаване на началните стойности на обект:

```

minchevAccount = new CustomerAccount("Dimitar Minchev",100);

```

В тази ситуация конструкторът в класа наследник ще трябва да извика определен конструктор в родителския клас, който да зададе началните настройки преди обекта да бъде създаден:

```

public CustomerAccount(string inName, string inAddress) :
base (inName, inAddress)
{
    ...
}

```

Ключовата дума **base** се използва по същия начин както **this** се

използва, за да извика друг конструктор в същия този клас.

Верижно свързване на конструктори

Когато обмисляте йерархии на конструктори и класове трябва следователно да запомните, че за да създадете инстанция на клас наследник първо трябва да е създадена инстанция на родителския клас. Това означава, че конструктора на родителя трябва първо да бъде изпълнен преди конструктора на наследника.

Резултата от това е, че програмистите трябва да се погрижат за проблема на **верижно свързване на конструктори**⁵².

Абстрактни методи и класове

C# ни предоставя начини за отбелязване на метод като **абстрактен**⁵³. Това означава, че тялото на метода не е осигурено в този клас, но ще бъде осигурено в класа наследник:

```
public abstract class Account
{
    public abstract string RudeLetterString();
}
```

Факта, че новият клас *Account* съдържа абстрактен метод означава, че самия клас е абстрактен (*и трябва да бъде маркиран като такъв*). Не е възможно да се направи инстанция на абстрактен клас.

Може да си мислим за абстрактния клас като вид шаблон. Ако

⁵²Верижно свързване на конструктори, тук се има предвид, че при създаване на обект от йерархично наследен клас, то се изпълнява първо конструктора на родителя и после конструктора на наследника. Термин на Английски език = **Constructor Chaining**, *вж.*

<http://www.codeproject.com/Articles/271582/Constructor-Chaining-in-Csharp>

⁵³Абстрактен означава, че нещото е незавършено и трябва да бъде реализирано в наследяващия клас. Термин на Английски език = **Abstract**, *вж.*

[http://msdn.microsoft.com/en-us/library/sf985hc5\(v=VS.71\).aspx](http://msdn.microsoft.com/en-us/library/sf985hc5(v=VS.71).aspx)

искате да направите инстанция на клас, базиран на абстрактен родител трябва да осигурите имплементации на всички абстрактни методи, които са дадени в родителския клас.

```
public interface IAccount
{
    void PayInFunds(decimal amount);
    bool WithdrawFunds(decimal amount);
    decimal GetBalance();
    string RudeLetterString();
}
public abstract class Account : IAccount
{
    private decimal balance = 0;
    public abstract string RudeLetterString();
    public virtual bool WithdrawFunds(decimal amount)
    {
        if(balance < amount)return false ;
        balance = balance-amount;
        return true;
    }
    public decimal GetBalance()
    {
        return balance;
    }
    public void PayInFunds(decimal amount)
    {
        balance = balance+amount;
    }
}
public class CustomerAccount : Account
{
    public override string RudeLetterString()
    {
        return "You are overdrawn";
    }
}
public class BabyAccount : Account
{
    public override bool WithdrawFunds(decimal amount)
    {
        if (amount > 10)return false;
        return base.WithdrawFunds(amount);
    }
    public override string RudeLetterString()
    {
        return "Tell daddy you are overdrawn";
    }
}
```


Референции към абстрактни класове

Референции към абстрактни класове работят точно като референциите към интерфейсите.

Проектиране на обекти и компоненти

Интерфейс - позволява да се идентифицират набор от поведения, които даден компонент може да имплементира. Всеки компонент, който имплементира даден интерфейс може да бъде разглеждан, като референция на този интерфейс. Обектите могат да бъдат имплементирани в повече от един интерфейс, позволявайки им да показват различни лица на системата, която ги използва.

Абстрактен - позволява да се създаде родителски клас, който да съдържа шаблон на информация за всички класове, които го наследяват. Ако искате да създадете комплект, например всички различни видове банкови сметки, може да ви се наложи да включите: кредитна карта, депозит, текуща сметка и др., тогава най-добрият начин да направим това е да създадем родителски клас, който да съдържа абстрактни и не абстрактни методи. Класът наследник може да използва методите от родителя и да припокрива тези, които трябва да се използват по различен начин за този определен клас.

Не изпадайте в паника!

Не изпадайте в паника⁵⁴. Всичко това са много дълбоки води. Тези функции на езика C# са обвързани с процесите на софтуерния дизайн, които са **много сложна работа**⁵⁵. Важното за

⁵⁴Игра на думи, „Не изпадайте в паника!“ е заглавието на книгата „Пътеводител на галактическия стопаджия“, съдържащ информацията за цялата галактика. Фраза на Английски език = **Don't Panic!**

⁵⁵Много сложна работа. Фраза в превод от Английски език = **very complex business**

запомняне е, че всички тези функции са ни осигурени, за да можем да решим всеки един софтуерен проблем.

Интерфейсите ни позволяват да опишем, какво всеки компонент може да направи. Йерархиите от класове ни позволяват да преизползваме програмен код в тези компоненти.

Обектите и методът ToString

Знаем, че един обект може да съдържа информация и да прави неща за нас. Всъщност до тук видяхме, че цялата основа на построяването на програми е да изберем, какво обектите трябва да правят и тогава да ги накараме да правят тези неща. Също така видяхме, че може да разширим родителски обект, за да създадем дете, което има всичките възможности и умения на родителя, плюс нови, които ние добавяме. Сега трябва да видим как тези умения на обектите се използват в направата на части от самите имплементации на C#, които работят.

Класът Обект

Когато създавате нов клас, той не просто се създава от нищото. Всъщност новия клас, който създавате е наследник на обектния клас *object*. С други думи, ако напиша:

```
public class Account {  
    // това е еквивалентно на написването на  
    public class Account : object {
```

Обектният клас е част от C# и всеки обект всъщност е наследник на обектния клас.

Метод ToString

Системата знае, че метод *ToString* съществува за всеки обект и ако някога се нуждае от низовата версия на обекта ще извика този метод за обекта, за да извлече съответния текст.

Хубавото на *ToString* е, че е деклариран като **virtual**. Това означава, че можем да го припокрием, за да го накараме да се държи както ние искаме:

```
class Account
{
    private string name;
    private decimal balance;

    public override string ToString()
    {
        return "Name: "+name+" balance: "+balance;
    }

    public Account(string inName, decimal inBalance)
    {
        name = inName;
        balance = inBalance;
    }
}
```

Припокрива метода *ToString*, така че той отпечатва името на притежателя и стойността на променливата съхраняваща баланса на сметката. Това означава, че програмният код:

```
Account a = new Account("Minchev",25);
Console.WriteLine(a);
```

ще отпечата:

```
Name: Minchev balance: 25
```

Обекти и тестване за равенство

В една игра множеството обекти ще бъдат разположени на определено място на екрана. Можем да изразим тази позиция като декартови координати или по-точно като точка в пространството с координати: стойност *x* и стойност *y*.

```
class Point
{
    public int x;
    public int y;
}
```

Това е моят клас точка

Забележете, че направих *x* и *y* членовете на класа *Point* публични. Това е така, защото не се интересувам от защитата им, а искам моята програма да работи по-бързо.

Добавяне на собствени методи за равенство

За да направим това трябва да припокрием стандартния метод за равенство и да добавим наш:

```
public override bool Equals(object obj)
{
    Point p = (Point) obj;
    if((p.x == x)&&(p.y == y))
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

На този метод за равенство му е подадена референцията към обекта, който искаме да сравним. Забележете, че референцията е доставена като референция към обект. Първото нещо, което трябва да направим е да създадем референция към класа *Point*. Трябва да направим това, защото искаме да се сдобием със стойностите *x* и *y* от класа *Point*.

Типа на обекта е каснат в тип *Point* и тогава се сравняват стойностите на *x* и *y*. Ако и двете са равни метода ще върне *true* към този, който го е извикал, в противен случай върнатата стойност ще бъде *false*. Това означава, че мога да напиша следния програмен код:

```
if(missilePosition.Equals(spaceshipPosition))
{
    Console.WriteLine("Bang");
}
```

Методът за равенство всъщност сравнява съдържанието на двете точки, вместо само на референциите към тях.

Обекти и this

Досега трябва да сте приели идеята да използвате референция към обект, за да се сдобие с членовете на този обект. Вижте следния програмен фрагмент:

```
public class Counter
{
    public int Data=0;

    public void Count()
    {
        Data = Data + 1;
    }
}
```

Класът по горе има единствена член данна и единствен член метод. Данната е брояч. Всеки път, когато метод *Count* бъде извикан, броячът нараства. Мога да използвам класа, както следва:

```
Counter c = new Counter();
c.Count();
Console.WriteLine("Count : "+c.Data);
```

Това извиква метода и след това отпечатва данните. Знаем, че в този контекст . (точката) означава: „*следвай референцията към обекта и след това използвай този член от класа*“.

this като референция към текущата инстанция

Запомнете, че когато използвам думата **this** всъщност имам в предвид: „*връзка към текущо изпълнящата се инстанция на клас*“. Когато метод в клас получи достъп до член променлива, компилатора автоматично слага **this**. Пред всяко използване на този член на класа.

```
public class Counter
{
    public int Data=0;

    public void Count()
    {
        this.Data = this.Data+1;
    }
}
```

Подаване на референция към себе си за други класове
Друга употреба на **this** е когато инстанция на клас трябва да осигури референция към себе си за друг клас, който иска да я използва.

```
bank.Store(this);
```

Когато методът *Store* е извикан, той получава референция към инстанцията, която се изпълнява в момента.

Силата на низовете и символите

Вероятно си струва да помислите известно време за типа на низа в детайли. Това е така, защото той ви дава много хубави допълнителни функции, които ще ви спестят много работа.

Сравняване на низове

Можете да сравните низове използвайки равенство:

```
if(s1 == s2)
{
    Console.WriteLine("The same");
}
```

или може да използвате метод *Equals*, ако предпочитате:

```
if(s1.Equals(s2))
{
    Console.WriteLine("Still the same");
}
```

Редактиране на низ

Можете да прочетете индивидуален символ от текстов низ като

го индексирате, както бихте направили това с масив. Например:

```
char firstch = name[0];
```

Можете да извадите серия символи от текстов низ използвайки метод *Substring*:

```
string s1="Minchev";  
s1=s1.Substring(1,2);
```

Можете да пропуснете втория параметър, ако желаете, тогава всички символи до края на низа ще бъдат копирани:

```
string s1="Minchev";  
s1=s1.Substring(2);
```

Дължина на низ

```
Console.WriteLine("Length: "+s1.Length);
```

Свойството *Length* връща броя на символи в текстов низ (*т.е. неговата дължина*).

Главни и малки букви

```
s1=s1.ToUpper();
```

Метод *ToUpper* връща версия на текстов низ, като всички букви са преобразувани в ГЛАВНИ. Другите символи не се променят. Има и съответен метод наречен *ToLower*, който преобразува текстовия низ в малки букви.

Премахване и празни низове

За премахване на всякакви водещи или завършващи интервали от низа, се използва метода:

```
s1=s1.Trim();
```

Съществуват също методи *TrimStart* и *TrimEnd*, които се използват за премахване на водещите или завършващите интервали в текстовия низ.

Символни команди

Класът *char* също има някои много полезни методи, които могат да бъдат използвани за да се проверят стойностите на индивидуални символи.

<code>char.IsDigit(ch)</code>	Връща „вярно“, ако символа е цифра (0 до 9)
<code>char.IsLetter(ch)</code>	Връща „вярно“, ако символа е буква (a до z или A до Z)
<code>char.IsLetterOrDigit(ch)</code>	Връща „вярно“, ако символа е буква или цифра
<code>char.IsLower(ch)</code>	Връща „вярно“, ако символа е малка буква
<code>char.IsUpper(ch)</code>	Връща „вярно“, ако символа е главна буква
<code>char.IsPunctuation(ch)</code>	Връща „вярно“, ако символа е пунктуационен символ
<code>char.IsWhiteSpace(ch)</code>	Връща „вярно“, ако символа е интервал, табулация или нов ред

Използвате тези методи при обхождане на текстов низ за търсене на определен символ в него.

Свойства

Свойствата са полезни. Това е!

Свойства като член клас

Свойство е член на клас, който съдържа стойност:

```
public class StaffMember
{
    public int Age;
}

StaffMember s = new StaffMember();
s.Age = 21;
```

Създаване на методите Get и Set

За да се сдобием с контрол и да извършваме полезни неща трябва да създадем методите *get* и *set*, които са публични. Те осигуряват достъп до член по управляван начин. Например:


```
public class StaffMember
{
    private int age;

    public int GetAge()
    {
        return this.age;
    }

    public void SetAge(int inAge)
    {
        if((inAge > 0)&&(inAge < 120))
        {
            this.age = inAge;
        }
    }
}
```

Вече имаме пълен контрол над нашите свойства, но трябваше да напишем доста допълнителен код. Ето пример и за тяхната употреба:

```
StaffMember s = new StaffMember();
s.SetAge(21);
Console.WriteLine("Age is: "+s.GetAge());
```

Използване на свойствата

Свойствата са начин, чрез който управлението на данните, е малко по – лесно.

```

public class StaffMember
{
    private int ageValue;
    public int Age
    {
        set
        {
            if((value > 0)&&(value < 120))
            {
                this.ageValue = value;
            }
        }
        get
        {
            return this.ageValue;
        }
    }
}

```

Стойността на променливата *age* сега е записана като свойство. Забележете как имаме *get* и *set* част към това конкретно свойството. Пример за употреба:

```

StaffMember s = new StaffMember();
s.Age = 21;
Console.WriteLine("Age is: "+s.Age);

```

Когато свойството *Age* получи стойност се изпълнява *set* кода. Когато свойството *Age* бъде прочетено се изпълнява *get* кода. Можем да пишем нови свойства:

```

public int AgeInMonths
{
    get
    {
        return this.ageValue*12;
    }
}

```

Това е ново свойство наречено *AgeInMonths*. Можете да осигурите свойства само за четене (*read-only*), като пропуснете *set* поведението. Свойства само за писане (*write-only*) са също възможни, ако пропуснете *get*.

Свойства и интерфейси

Интерфейсите са начин, по който можете да съберете набор от поведения. Също така е възможно да добавите свойства към интерфейсите. Например:

```
interface IStaff
{
    int Age
    {
        get;
        set;
    }
}
```

Използване на Делегати

Делегатите⁵⁶ са важна част от това как събитията се управляват. Събитията са неща, които се случват и на които нашата програма трябва да отговори. Те включват например: натискане на бутон в потребителския интерфейс, часовници, които тикат или съобщения пристигащи по мрежата. Във всеки един случай трябва да казваме на системата какво да прави, когато се случи събитие. Начинът, по който С# прави това е като ни позволява да създаваме инстанции на класове делегати, на които даваме генератори на събитията. Когато събитието се случи (*понякога това се казва още събитието беше „изстреляно“*) методът, който е посочен от събитието бива извикан, за да достави съобщение. Един делегат наричаме: *„Начин, по който да кажем на парче от програма, какво да прави, когато нещо се случи.“*

Ако извикате делегата, той извиква метода, който го реферира в момента. Това означава, че мога да го използвам като избор на метод.

⁵⁶Термин на Английски език = **Delegates**, вж. <http://msdn.microsoft.com/en-us/library/vstudio/ms173171.aspx>

```
public delegate decimal CalculateFee(decimal balance);
```

Забележете, че не съм създал никакви делегати все още, просто казах на компилатора как типа делегат *CalculateFee* изглежда.

Пример за метод, който може да искаме да използваме с този делегат е кода:

```
public decimal RipoffFee(decimal balance)
{
    if(balance < 0) return 100;
    else return 1;
}
```

Ако искам да използвам това в моята програма, мога да направя инстанция на *CalculateFee*, която прави връзка към него по следния начин:

```
CalculateFee calc = new CalculateFee(RipoffFee);
```

Сега мога да „извикам“ този делегат и той всъщност ще пусне съкратен метод на калкулатор:

```
fees = calc(100);
```

В момента делегата *calc* се отнася към делегата инстанция, която ще използва метода *RipoffFee*. Мога да променя това като направя друга инстанция на делегата по следния начин:

```
calc = new CalculateFee(FriendlyFee);
```

Това ни дава още един слой на абстракция и означава, че сега можем да проектираме програми, които променят своето поведение докато програмата работи.

Съвет: Използвайте делегатите благоразумно

Делегатите се използват много при обработката на събития, а също и за да управляват нишки.

ГЛАВА 5. ПРОГРАМИРАНЕ ЗА НАПРЕДНАЛИ

Настоящата глава засяга програмиране за напреднали. Разгледана е структурата данни списък и нейната употреба. Запознаване с възможностите за многозадачна работа на процесора посредством употребата на нишки, включващо: добавяне на поддръжката на нишки; създаване и стартиране на нишка; създаване на повече нишки; синхронизация на нишките; контрол на нишки; намиране на състоянието на нишка.

Списък

В случай на елементи с общи признаци, можете да използвате класа *List*. При създаването на списък имате начин да укажете какъв тип данни ще се съхранява в него. Функциите на C#, които се грижат за еднаквите елементи (*елементите с общи признаци*) добавят няколко нови нотации, за да ви позволят да изразите това. Класът за списък *List* е част от пространството от имена *System.Collections.Generic* и работи по следния начин:

```
List<Account> accountList = new List<Account>();
```

Горната програмна конструкция създава списък наречен *accountList*, който съдържа референции към *Accounts*. Типът между символите < и > указва как да кажем на списъка какъв тип данни искаме той да съдържа.

Примерна програма обработваща списък е представена на фрагмента по-долу:

```

using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        List<int> list = new List<int>();
        list.Add(1);
        list.Add(2);
        list.Add(3);

        // печат на елементи от списъка
        foreach (int element in list)
            Console.WriteLine(element);

        // премахване на елементи от списъка
        foreach (int element in list)
            list.Remove(element);
    }
}

```

Нишки

Ако искате да се наричате истински програмист трябва да знаете няколко неща за нишките.

Какво е нишка?

Досега нашите програми използваха само една **нишка**⁵⁷, по време на своята работа. Нишката обикновено започва в метода *Main* и свършва, когато се достигне края на този метод. Можете да си мислите за една нишка като за влак, който се движи по релси. Релсите са програмните конструкции, които нишката изпълнява. По същия начин, по който можете да сложите повече от един влак на релсите, така и компютърът може да изпълнява повече от една нишка в един блок програмен код.

⁵⁷Нишките позволяват едновременна обработка, т.е. извършва се повече от една операция в даден момент. Термин от Английски език = **threads, threading**, вж. <http://msdn.microsoft.com/en-us/library/ms173178.aspx>

Защо имаме нишки?

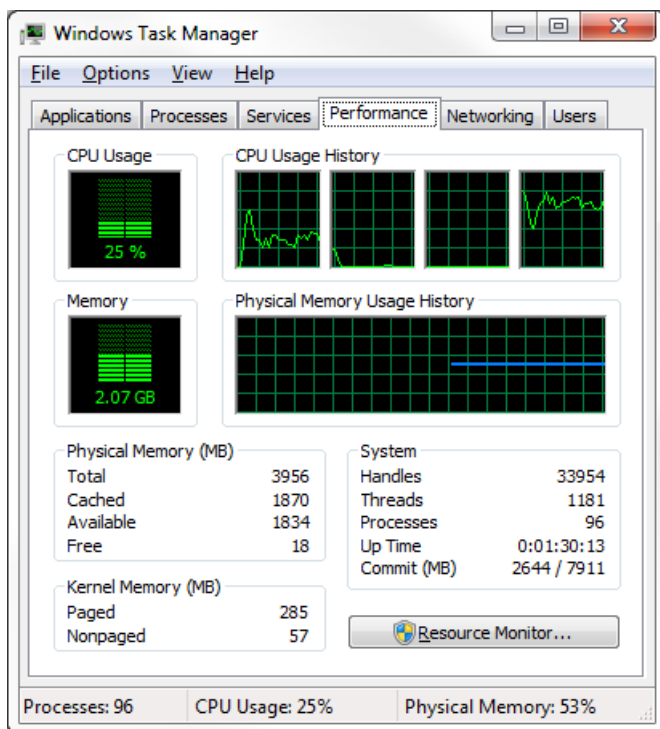
Нишките предоставят друго ниво на абстракция, в смисъл, че ако искаме да направим повече от едно нещо по едно и също време, е много по – ползотворно да изпратим нишка, която да изпълни задачата, вместо да се пробваме да преплитаме втора задача с първата.

Нишките и процесора

Разгледайте следния метод представен в програмния фрагмент по-долу:

```
static private void BusyLoop()
{
    long count;
    for(count=0;count<10000000000000L;count=count+1)
    {
        ;;
    }
}
```

Този метод не прави нищо, но го прави много милиона пъти. Ако стартирате *Task Manager* ще видите:



Моят лаптоп има 4 процесорни ядра, което означава, че мога да изпълня четири нишки едновременно. Можете да видите малките графи под централния процесор (CPU). Частта *Usage History* на дисплея показва колко е ангажиран всеки един процесор. На фигурата се вижда, че най – левия и най – десния процесор са доста заети, но средните два процесора не правят абсолютно нищо.

Полезен съвет: Повече нишки могат да подобрят производителността

Ако можете да направите така, че Вашата програма да се разпростира на няколко нишки, това може да окаже голямо влияние на скоростта, с която тя работи.

Добавяне на поддръжката на нишки в програмата
Най-лесният начин да се уверим, че можем да използваме ресурса нишки, е да добавим тяхната поддръжка, чрез пространството от имена *System.Threading* в началото на нашата програма, например ето така:

```
using System.Threading;
```

Класът *Thread* предоставя връзка между програмата и операционната система.

Укажете къде започва изпълнението на нишка
Когато създадете нишка Вие трябва да укажете, къде тя да започне да се изпълнява. Можете да направите това използвайки делегати. Както вече знаем, делегат е начин на рефериране към метод от клас. Типът делегат използван от нишката може да се отнася до метод, който не връща стойност или приема каквито и да е параметри.

```
ThreadStart busyLoopMethod = new ThreadStart(BusyLoop);
```

Създаване на нишка

Веднъж след като сте указали къде започва изпълнението на нишка, можете да създадете нова нишка:

```
Thread t1 = new Thread(busyLoopMethod);
```

Променливата *t1* сега реферира нова инстанция на нишка. Забележете в момента нишката не се изпълнява, вместо това тя чака да бъде стартирана. Това ще направим сега!

Стартиране на нишка

Класът нишка ни снабдява с няколко метода, които Вашата програма може да използва, за да контролира различните

процеси и тяхното изпълнение. За да стартирате нишка може да използвате например, метода *Start* по следния начин:

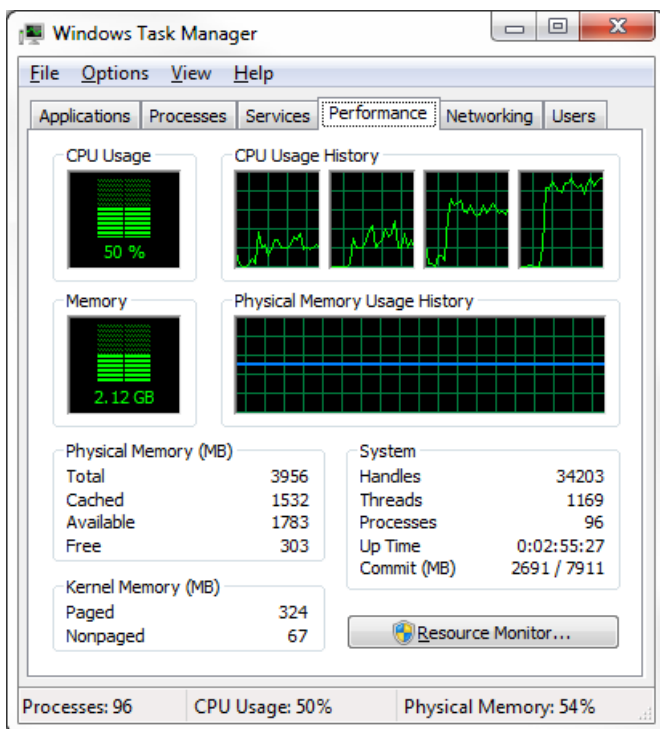
```
t1.Start();
```

Това е момента, в който нишката започва да се изпълнява. Програмният фрагмент по-долу съгласява цялата програма демонстрираща употребата на нишки:

```
using System;
using System.Threading;

class ThreadDemo
{
    static private void BusyLoop()
    {
        long count;
        for(count=0;count<1000000000000L;count=count+1)
        {
        }
    }

    static void Main()
    {
        ThreadStart busyLoopMethod = new
ThreadStart(BusyLoop);
        Thread t1 = new Thread(busyLoopMethod);
        t1.Start();
        BusyLoop();
    }
}
```



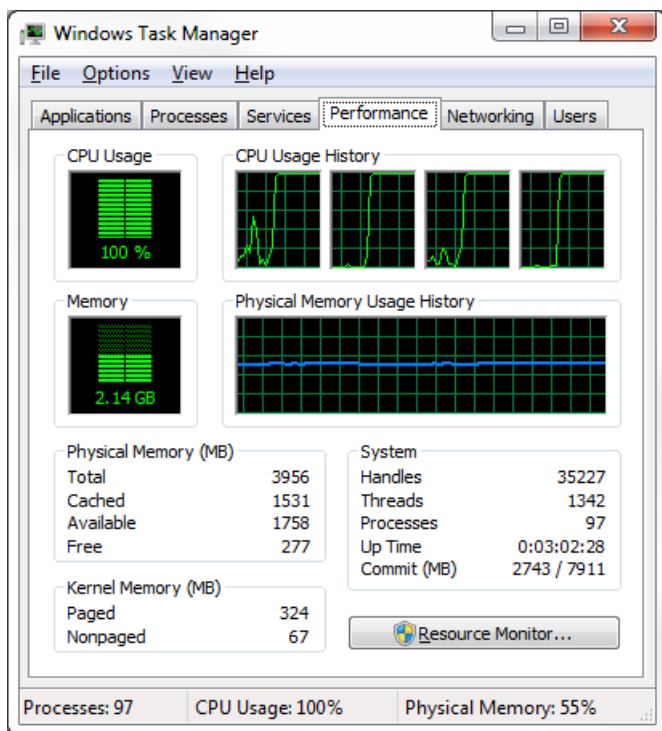
На фигурата виждате, че сега имаме по-голяма активност и употребата на процесора се е повишила от 25% на 50%, понеже нашата програма сега използва нишки.

Създаване на повече нишки

Нашият пример показва, че като създадем допълнителни нишки, ние използваме повече от процесорната мощ на компютъра. Всъщност, можем да създадем много повече нишки като тази:

```
for(int i=0;i<100;i=i+1)
{
    Thread t1 = new Thread(busyLoopMethod);
    t1.Start();
}
```

Този цикъл ще създаде 100 нишки, като всяка една от тях изпълнява метода *BusyLoop*. Сега нашият компютър е наистина зает, както ще видите на фигурата по-долу:



Всичките процесори сега работят на максимум и използването на процесора е на 100%. Ако изпълните този код, всъщност може да установите, че машината ви е станала малко по – бавна и че всички охлаждащи вентилатори работят с пълна сила.

Полезен съвет: Прекалено много нишки ще забавят всичко

Забележете как нищо не спря Вашата програма от стартирането на голям брой работещи нишки. Това е потенциално опасно! Ако стартирате прекалено много нишки наведнъж само ще забавите всичко. Такава е всъщност основата на атака към компютъра наречена - **Denial of Service Attack** (атака от тип отказ на достъп и/или услуга).

Нишки и синхронизация

Можем да направим малка промяна на този код и да направим променливата *count* член на класа:

```
static long count;
static private void BusyLoop()
{
    for(count=0;count<10000000000000L;count=count+1)
    {
    }
}
```

Този метод изглежда доста подобен, но сега всяка нишка изпълняваща *BusyLoop* споделя една и съща променлива *count*. Потенциално това е катастрофа! Сега е много трудно да се прецени колко време ще отнеме на тази програма с множество нишки да приключи успешно своята работа.

Използване на взаимно изключване за управление на споделени данни

Взаимното изключване работи по абсолютно същия начин, както и използването на инстанция на обект, който да играе ролята на контролър.

```
static object sync = new object();
```

Този обект всъщност не съдържа никакви данни. Просто се използва за контролър, който се съдържа в активния процес.

```
Monitor.Enter(sync);  
count = count+1;  
Monitor.Exit(sync);
```

Кодът между извикването на *Monitor.Enter* и *Monitor.Exit* може единствено да бъде изпълнен от една нишка по в даден момент. С други думи, няма опасност *Windows* да преустанови да интерпретира програмната конструкция инкрементация и да превключи на друг процес. Всичките операции за инкрементиране ще завършат изцяло. Когато изпълнението напусне програмните конструкции между извикванията на *Monitor*, нишката може да бъде спряна както обикновено.

Нишките, които не могат да бъдат пуснати са „паркирани“ в опашка от чакащи нишки. Тази опашка се пази в ред, така че първата нишка, която стигне до входа на опашката, ще бъде първата, която ще вземе жетона, докато другите нишки наредени чакат своя ред. Класът *Monitor* се грижи вместо нас за всичко това и не е нужно да знаем как работи.

Полезен съвет: Нишките могат абсолютно да счупят Вашата програма

Ако нишка заеме синхронизиращ обект и не го пусне, това ще спре изпълнението на другите нишки, които се нуждаят от този обект. Това е наистина добър начин да провалите Вашите програми. Друг такъв начин, наречен „Мъртва хватка“ се получава, когато нишка има обект x и чака за обект y, а нишка b има обект y и чака за обект x. Това е компютърната версия на „Няма да му се обадя, за да се извиня, ще изчакам той да ми се обади.“

Контрол на нишки

Има няколко допълнителни особености, които можем да използваме за управление на нишки.

Паузиране на нишки

```
Thread.Sleep(500);
```

Този метод получава като параметър броя **милисекунди**⁵⁸, който изпълнението на нишка трябва да бъде преустановено. Горното извикване например би паузирало нишка в програмата за половин секунда.

Свързване на нишки

Може да поискате една нишка да изчака друга нишка да свърши да работи:

```
t1.Join();
```

Това ще накара изпълняваната текуща нишка да изчака, докато нишката *t1* не приключи своята работа.

Контрол на нишки

Класът *Thread* ни осигурява няколко метода, които можем да използваме, за да контролираме изпълнението на нишка, по следния начин:

```
t1.Abort();
```

Това кара операционната система да унищожи тази нишка и да я изтрие от паметта.

Ако не искате да унищожите нишка, но просто да я паузирате за малко, можете да използвате метода *Suspend* за тази нишка, по следния начин:

```
t1.Suspend();
```

⁵⁸Милисекундата е една хилядна от секундата.

Нишката е приспана, докато не извикате метода *Resume* за нея, по следния начин:

```
t1.Resume();
```

Намиране на състоянието на нишка

Можете да намерите състоянието на нишка използвайки нейното свойство *ThreadState*. Това е енумерирана стойност, която има няколко възможни стойности, като най – често използваните от тях са:

<i>ThreadState.Running</i>	нишката се изпълнява
<i>ThreadState.Stopped</i>	нишката е спряла да изпълнява нишковия метод
<i>ThreadState.Suspend</i>	нишката е била приспана
<i>ThreadState.WaitSleepJoin</i>	нишката изпълнява <i>Sleep</i> , чака да бъде свързана или чака за <i>Monitor</i> .

Следващия програмен фрагмент е пример, който показва съобщение, дали нишката *t1* се изпълнява:

```
if(t1.ThreadState == ThreadState.Running)
{
    Console.WriteLine("Thread Running");
}
```


ИЗТОЧНИЦИ

1. C# Yellow Book, <http://www.csharpcourse.com/>
2. Rob Miles, <http://www.robmiles.com>
3. Microsoft Developer Network, <http://msdn.microsoft.com>
4. Microsoft Visual Studio,
<http://www.microsoft.com/visualstudio/>
5. Wikipedia, English, <http://en.wikipedia.org/wiki/>
6. Code Project, <http://www.codeproject.com>
7. Stack Overflow, <http://www.stackoverflow.com/>
8. Телерик Академия, <http://academy.telerik.com>
9. Светлин Наков, <http://www.nakov.com>
10. Димитър Минчев, <http://www.minchev.eu>

© Димитър Минчев

Жълта книга по C#

Издателство Божич” Бургас, 2013

bojich__bg@abv.bg

ISBN 978-954-9925-84-5