**Класове и обекти**

През последните няколко десетилетия програмирането и информатиката като цяло претърпяват невероятно развитие и се появяват концепции, които променят изцяло начина, по който се изграждат програми. Точно такава радикална идея въвежда **обектно-ориентираното програми­ране (ООП)**. Ще изложим кратко въведение в принципите на ООП и понятията, които се използват в него. Като начало ще обясним какво представляват класовете и обектите. Тези две понятия стоят в основата на ООП и са неразделна част от ежедне­вието на почти всеки съвременен програмист.

**Какво е обектно-ориентирано програмиране?**

Обектно-ориентираното програмиране е модел на програмиране, който използва **обекти** и техните взаимодействия за изграждането на ком­пютърни програми. По този начин се постига лесен за разбиране, опростен модел на предметната област, който дава възможност на програмиста интуитивно (чрез проста логика) да решава много от задачите, които възникват в реалния свят.

Засега няма да навлизаме в детайли за това какви са целите и предимствата на ООП, както и да обясняваме подробно принци­пите при изграждане на йерархии от класове и обекти. Ще вмъкнем само, че програмните техники на ООП често включват капсулация, модулност, полиморфизъм и наследяване. Тези техники са извън целите на настоящата тема, затова ще ги разгледаме по-късно в главата "[Принципи на обектно-ориентира­ното програмиране](http://www.introprogramming.info/intro-csharp-book/read-online/glava20-principi-na-obektno-orientiranoto-programirane)". Сега ще се спрем на обектите като основно понятие в ООП.

**Какво е обект?**

Ще въведем понятието **обект** в контекста на ООП. Софтуерните обекти моделират обекти от реалния свят или абстрактни концепции (които също разглеждаме като обекти).

Примери за реални обекти са хора, коли, стоки, покупки и т.н. Абстракт­ните обекти са понятия в някоя предметна област, които се налага да моделираме и използваме в компютърна програма. Примери за абстрактни обекти са структурите от данни стек, опашка, списък и дърво. Те не са предмет на настоящата тема, но ще ги разгледаме в детайли в следващите теми.

В обектите от реалния свят (също и в абстрактните обекти) могат да се отделят следните две групи техни характеристики:

-     Състояния (states) – това са характеристики на обекта, които по някакъв начин го определят и описват по принцип или в конкретен момент.

-     Поведения (behaviors) – това са специфични характерни действия, които обектът може да извършва.

Нека за пример вземем обектът от реалния свят "куче". Състояния на кучето могат да бъдат "име", "цвят на козината" и "порода", а негови поведения – "лаене", "седене" и "ходене".

Обектите в ООП обединяват данни и средствата за тяхната обработка в едно цяло. Те съответстват на обектите от реалния свят и съдържат в себе си данни и действия:

-     Член-данни (data members) – представляват про­менливи, вградени в обектите, които описват състоянията им.

-     Методи (methods) – вече сме ги разглеждали в детайли. Те са инструментът за изграждане на поведението на обектите.

**Какво е клас?**

**Класът** дефинира абстрактните характеристики на даден обект. Той е план или шаблон, чрез който се описва природата на нещо (някакъв обект). Класовете са градивните елементи на ООП и са неразделно свързани с обектите. Нещо повече, всеки обект е пред­ставител на точно един клас.

Ще дадем пример за клас и обект, който е негов представител. Нека имаме клас **Dog** и обект **Lassie**, който е представител на класа **Dog** (казваме още обект от тип **Dog).** Класът **Dog** описва характеристиките на всички кучета, докато **Lassie** е конкретно куче**.**

Класовете предоставят модулност и структурност на обектно-ориентира­ните програми. Техните характеристики трябва да са смислени в общ контекст, така че да могат да бъдат разбрани и от хора, които са запознати с проблемната област, без да са програмисти. Например, не може класът **Dog** да има характеристика "RAM памет" поради простата причина, че в контекста на този клас такава характеристика няма смисъл.

**Класове, атрибути и поведение**

Класът дефинира характеристиките на даден обект (които ще наричаме **атрибути**) и неговото **поведение** (действията, които обектът може да извършва). Атрибутите на класа се дефинират като собствени променливи в тялото му (наречени член-променливи). Поведението на обектите се моделира чрез дефиниция на методи в класовете.

Ще илюстрираме казаното дотук като дадем пример за реална дефиниция на клас. Нека се върнем отново на примера с кучето, който вече дадохме по-горе. Искаме да дефинираме клас **Dog**, който моделира реалният обект "куче". Класът ще включва характеристики, общи за всички кучета (като порода и цвят на козината), а също и характерно за кучетата поведение (като лаене, седене, ходене). В такъв случай ще имаме атрибути **breed** и **furColor**, а поведението ще бъде имплементирано чрез методите **Bark()**, **Sit()** и **Walk()**.

**Обектите – инстанции на класовете**

От казаното дотук знаем, че всеки обект е представител на точно един клас и е създаден по шаблона на този клас. Създаването на обект от вече дефиниран клас наричаме **инстанциране** (instantiation). **Инстанция** (instance) е фактическият обект, който се създава от класа по време на изпълнение на програмата.

Всеки обект е **инстанция** на конкретен клас. Тази инстанция се характе­ризира със **състояние** (state) – множество от стойности, асоциирани с атрибутите на класа.

В контекста на така въведените понятия, обектът се състои от две неща: моментното състояние и поведението, дефинирано в класа на обекта. Състоянието е специфично за инстанцията (обекта), но поведението е общо за всички обекти, които са представители на този клас.

**Класове в C#**

До момента разгледахме някои общи характеристики на ООП. Голяма част от съвременните езици за програмиране са обектно-ориентирани. Всеки от тях има известни особености при работата с класове и обекти. В настоящата книга ще се спрем само на един от тези езици – C#. Хубаво е да знаем, че знанията за ООП в C# ще бъдат от полза на читателя без значение кой обектно-ориентиран език използва в практиката, тъй като ООП е фундаментална концепция в програмирането, използвана от почти всички съвременни езици за програмиране.

**Какво представляват класовете в C#?**

Класът в C# се дефинира чрез ключовата дума **class**, последвана от идентификатор (име) на класа и съвкупност от член-данни и методи, обособени в собствен блок код.

Класовете в C# могат да съдържат следните елементи:

-     Полета (fields) – член-променливи от определен тип;

-     Свойства (properties) – това са специален вид елементи, които разширяват функционалността на полетата като дават възможност за допълнителна обработка на данните при извличането и записва­нето им в полетата от класа. Ще се спрем по-подробно на тях в темата "[Дефиниране на класове](http://www.introprogramming.info/intro-csharp-book/read-online/glava14-definirane-na-klasove)";

-     Методи – реализират манипулацията на данните.

**Примерен клас**

Ще дадем пример за прост клас в C#, който съдържа изброените елементи. Класът **Cat** моделира реалния обект "котка" и притежава свой­ствата име и цвят. Посоченият клас дефинира няколко полета, свойства и методи, които по-късно ще използваме наготово. Следва дефиницията на класа (засега няма да разглеждаме в детайли дефиницията на класо­вете – ще обърнем специално внимание на това в главата "[Дефиниране на класове](http://www.introprogramming.info/intro-csharp-book/read-online/glava14-definirane-na-klasove)"):

|  |
| --- |
| public class Cat{ // Field name private string name; // Field color private string color;  public string Name { // Getter of the property "Name" get { return this.name; } // Setter of the property "Name" set { this.name = value; } }  public string Color { // Getter of the property "Color" get { return this.color; } // Setter of the property "Color" set { this.color = value; } }  // Default constructor public Cat() { this.name = "Unnamed"; this.color = "gray"; }  // Constructor with parameters public Cat(string name, string color) { this.name = name; this.color = color; }  // Method SayMiau public void SayMiau() { Console.WriteLine("Cat {0} said: Miauuuuuu!", name); }} |

Примерният клас **Cat** дефинира свойствата **Name** и **Color**, които пазят стойността си в скритите (private) полетата **name** и **color**. Допълнително са дефинирани два конструктора за създаване на инстанции от класа **Cat** съответно без и със параметри и метод на класа **SayMiau()**.

След като примерният клас е дефиниран, можем вече да го използваме, например по следния начин:

|  |
| --- |
| static void Main(){ Cat firstCat = new Cat(); firstCat.Name = "Tony"; firstCat.SayMiau();  Cat secondCat = new Cat("Pepy", "Red"); secondCat.SayMiau(); Console.WriteLine("Cat {0} is {1}.", secondCat.Name, secondCat.Color);} |

Ако изпълним примера, ще получим следния резултат:

|  |
| --- |
| Cat Tony said: Miauuuuuu!Cat Pepy said: Miauuuuuu!Cat Pepy is Red. |

Видяхме прост пример за дефиниране и използване на класове, а в секцията "[Създаване и използване на обекти](http://www.introprogramming.info/intro-csharp-book/read-online/glava11-sazdavane-i-izpolzvane-na-obekti/%22%20%5Cl%20%22_%D0%A1%D1%8A%D0%B7%D0%B4%D0%B0%D0%B2%D0%B0%D0%BD%D0%B5_%D0%B8_%D0%B8%D0%B7%D0%BF%D0%BE%D0%BB%D0%B7%D0%B2%D0%B0%D0%BD%D0%B5)" ще обясним в подробности как се създават обекти, как се достъпват свойствата им и как се извикват методите им и това ще ни позволи да разберем как точно работи примерът.

**Системни класове**

Извикването на метода **Console.WriteLine(…)** на класа **System.Console** е пример за употребата на **системен клас** в C#. Системни наричаме класовете, дефинирани в стандартните библиотеки за изграждане на приложения със C# (или друг език за програмиране). Те могат да се използват във всички наши .NET приложения (в частност тези, които са написани на C#). Такива са например класовете **String**, **Environment** и **Math**, които ще разгледаме малко по-късно.

Важно е да се знае, че имплементацията на логиката в класовете е **капсулирана** (скрита) вътре в тях. За програмиста е от значение какво правят методите, а не как го правят и за това голяма част от класовете не е публично достъпна (**public**). При системните класове имплемента­цията често пъти дори изобщо не е достъпна за програмиста. По този начин се създават **нива на абстракция**, което е един от основните принципи в ООП.

Ще обърнем специално внимание на системните класове малко по-късно. Сега е време да се запознаем със създаването и използването на обекти в програмите.

**Създаване и използване на обекти**

Засега ще се фокусираме върху създаването и използването на обекти в нашите програми. Ще работим с вече дефинирани класове и ней-вече със системните класове от .NET Framework. Особеностите при дефинирането на наши соб­ствени класове ще разгле­даме по-късно в темата "[Дефини­ране на класове](http://www.introprogramming.info/intro-csharp-book/read-online/glava14-definirane-na-klasove)".

**Създаване и освобождаване на обекти**

Създаването на обекти от предварително дефинирани класове по време на изпълнението на програмата става чрез оператора **new**. Новосъз­даденият обект обикновено се присвоява на променлива от тип, съвпадащ с класа на обекта (това, обаче, не е задължително - вижте глава "[Принципи на обектно-ориентираното програмиране](http://www.introprogramming.info/intro-csharp-book/read-online/glava20-principi-na-obektno-orientiranoto-programirane)"). Ще отбележим, че при това присвояване същинският обект не се копира, а в променливата се записва само **референция** към новосъздадения обект (неговият адрес в паметта). Следва прост пример как става това:

|  |
| --- |
| Cat someCat = new Cat(); |

На променливата **someCat** от тип **Cat** присвояваме новосъздадена инстан­ция на класа **Cat**. Променливата **someCat** стои в стека, а нейната стойност (инстанцията на класа **Cat**) стои в динамичната памет (managed heap):



**Създаване на обекти със задаване на параметри**

Сега ще разгледаме леко променен вариант на горния пример, при който задаваме параметри при създаването на обекта:

|  |
| --- |
| Cat someCat = new Cat("Johnny", "brown"); |

В този случай искаме обектът **someCat** да представлява котка, която се казва **"Johnny"** и има кафяв цвят. Указваме това чрез думите **"Johnny"** и **"brown"**, написани в скоби след името на класа.

При създаването на обект с оператора **new** се случват две неща**:** заделя се памет за този обект и се извършва начална инициализация на член-данните му. Инициализацията се осъществява от специален метод на класа, наречен **конструктор**. В горния пример инициализиращите пара­метри са всъщност параметри на конструктора на класа. Ще се спрем по-подробно на конструкторите след малко. Понеже член-променливите **name** и **color** на класа **Cat** са от референтен тип (от класа **String**), те се записват също в динамичната памет (heap) и в самия обект стоят техните референции (адреси). Следващата картинка показва това нагледно:



**Освобождаване на обектите**

Важна особеност на работата с обекти в C# e, че обикновено няма нужда от ръчното им разрушаване и освобождаване на паметта, заета от тях. Това е възможно поради вградената в .NET CLR система за почистване на паметта (**garbage collector)**, която се грижи за освобождаването на неизползвани обекти вместо нас. Обектите, към които в даден момент вече няма референция в програмата, автоматично се унищожават и паметта, която заемат се освобождава. По този начин се предотвратяват много потенциални бъгове и проблеми. Ако искаме ръчно да освободим даден обект, трябва да унищожим референцията към него, например така:

|  |
| --- |
| someCat = null; |

Това не унищожава обекта веднага, но го оставя в състояние, в което той е недостъпен от програмата и при следващото включване на системата за почистване на паметта (garbage collector), той ще бъде освободен:



**Достъп до полета на обекта**

Достъпът до полетата и свойствата (properties) на даден обект става чрез оператора **.** (точка), поставен между името на обекта и името на полето (или свойството). Операторът **.** не е необходим в случай, че достъпваме поле или свойство на даден клас в тялото на метод от същия клас.

Можем да достъпваме полетата и свойствата или с цел да извлечем данните от тях, или с цел да запишем нови данни. В случай на свойство, достъпът се реализира по абсолютно същия начин както и при поле – C# ни предоставя тази възможност. Това се постига чрез двете специални ключови думи **get** и **set** в дефиницията на свойството, които извършват съответно извличането на стойността на свойството и присвояването на нова стойност. В дефиницията на класа **Cat** (която дадохме по-горе) свойства са **Name** и **Color**.

**Достъп до полета и свойства на обект – пример**

Ще дадем прост пример за употребата на свойство на обект, като използваме вече дефинирания по-горе клас **Cat**. Създаваме инстанция **myCat** на класа **Cat** и присвояваме стойност **"Alfred"** на свойството **Name.** След това извеждаме на стандартния изход форматиран низ с името на нашата котка. Следва реализацията на примера:

|  |
| --- |
| class CatManipulating{ static void Main() { Cat myCat = new Cat(); myCat.Name = "Alfred";  Console.WriteLine("The name of my cat is {0}.", myCat.Name); }} |

**Извикване на методи на обект**

Извикването на методите на даден обект става отново чрез оператора **.** (точка). Операторът точка не е нужен единствено, в случай че съответ­ният метод се извиква в тялото на друг метод от същия клас.

Сега е моментът да споменем факта, че методите на класовете имат **модификатори за достъп public**, **private** или **protected**, чрез които възможността за извикването им може да се ограничава. Ще разгледаме подробно тези модификатори в темата "[Дефиниране на класове](http://www.introprogramming.info/intro-csharp-book/read-online/glava14-definirane-na-klasove)". Засега е достатъчно да знаем само, че модификаторът за достъп **public** не въвежда никакво ограничение за извикването на съответния метод, т.е. го прави публично достъпен.

**Извикване на методи на обект – пример**

Ще допълним примера, който вече дадохме като извикаме метода **SayMiau** на класа **Cat**. Ето какво се получава:

|  |
| --- |
| class CatManipulating{ static void Main() { Cat myCat = new Cat(); myCat.Name = "Alfred";  Console.WriteLine("The name of my cat is {0}.", myCat.Name); myCat.SayMiau(); }} |

След изпълнението на горната програма на стандартния изход ще бъде изведен следният текст:

|  |
| --- |
| The name of my cat is Alfred.Cat Alfred said: Miauuuuuu! |

**Конструктори**

Конструкторът е специален метод на класа, който се извиква автоматично при създаването на обект от този клас и извършва инициализация на данните му (това е неговото основно предназначение). Конструкторът няма тип на връщана стойност и неговото име не е произволно, а задължително съвпада с името на класа. Конструкторът може да бъде със или без параметри.

Конструктор без параметри наричаме още конструктор по подразбиране (default constructor). Езикът C# допуска да няма изрично дефиниран конструктор в класа. В този случай, компилаторът създава автоматично празен конструктор по подразбиране, който занулява всички полета на класа.

**Конструктори с параметри**

Конструкторът може да приема параметри, както всеки друг метод. Всеки клас може да има произволен брой конструктори с единственото ограни­чение, че броят и типът на параметрите им трябва да бъдат различни. При създаването на обект от този клас се извиква точно един от дефинираните конструктори.

При наличието на няколко конструктора в един клас естествено възниква въпросът кой от тях се извиква при създаването на обект. Този проблем се решава по много интуитивен начин, както при методите. Подходящият конструктор се избира автоматично от компилатора в зависимост от пода­дената съвкупност от параметри при създава­нето на обекта. Използва се принципът на най-добро съвпадение.

**Извикване на конструктори – пример**

Да разгледаме отново дефиницията на класа **Cat** и по-конкретно двата конструктора на класа:

|  |
| --- |
| public class Cat{ // Field name private string name; // Field color private string color;  …  // Default constructor public Cat() { this.name = "Unnamed"; this.color = "gray"; }  // Constructor with parameters public Cat(string name, string color) { this.name = name; this.color = color; }  …} |

Ще използваме тези конструктори, за да илюстри­раме употребата на конструктор без и с параметри. При така дефинирания клас **Cat** ще дадем пример за създаването на негови инстанции чрез всеки от двата конструк­тора. Единият обект ще бъде обикновена неопределена котка, а другият – нашата кафява котка Johnny. След това ще изпълним метода **SayMiau** на всяка от двете и ще разгледаме резултата. Следва изходният код:

|  |
| --- |
| class CatManipulating{ static void Main() { Cat someCat = new Cat();  someCat.SayMiau(); Console.WriteLine("The color of cat {0} is {1}.", someCat.Name, someCat.Color);  Cat someCat = new Cat("Johnny", "brown");  someCat.SayMiau(); Console.WriteLine("The color of cat {0} is {1}.", someCat.Name, someCat.Color); }} |

В резултат от изпълнението на програмата се извежда следният текст на стандартния изход:

|  |
| --- |
| Cat Unnamed said: Miauuuuuu!The color of cat Unnamed is gray.Cat Johnny said: Miauuuuuu!The color of cat Johnny is brown. |

**Статични полета и методи**

Член-данните, които разглеждахме досега, реализират състояния на обектите и са пряко свързани с конкретни инстанции на класовете. В ООП има специална категория полета и методи, които се асоциират с тип данни (клас), а не с конкретна негова инстанция (обект). Наричаме ги **статични членове** (static members), защото са независими от конкретните обекти. Нещо повече, те се използват без да има създадена инстанция на класа, в който са дефинирани. Те могат да бъдат полета, методи и конструктори. Да разгледаме накратко статичните членове в C#.

**Статично поле или метод** в даден клас се дефинира чрез ключовата дума **static**, пос­тавена преди типа на полето или типа на връщаната стойност на метода. При дефинирането на статичен конструктор думата **static** се поставя преди името на конструктора. Статичните конструктори не са предмет на настоящата тема – засега ще се спрем само на статич­ните полета и методи (по-любознателните читатели могат да направят справка в MSDN).

**Кога да използваме статични полета и методи?**

За да отговорим на този въпрос трябва преди всичко добре да разбираме разликата между статичните и нестатичните (non-static) членове. Ще раз­гледаме по-детайлно каква е тя.

Вече обяснихме основната разлика между двата вида членове. Нека интерпретираме класа като категория обекти, а обектът – като елемент, представител на тази категория. Тогава статичните членове отразяват състояния и поведе­ния на самата категория обекти, а нестатичните – състояния и поведения на отделните представители на категорията.

Сега ще обърнем по-специално внимание на инициализацията на статич­ните и нестатичните полета. Вече знаем, че нестатичните полета се ини­циализират заедно с извикването на конструктор на класа при създа­ването на негова инстанция – или в тялото на конструктора, или извън него. Инициализацията на статичните полета, обаче, не може да става при създаването на обект от класа, защото те могат да бъдат използвани, без да има създадена инстанция на този клас. Важно е да се знае следното:

|  |  |
| --- | --- |
| clip_image007 | **Статичните полета се инициализират, когато типът данни (класът) се използва за пръв път по време на изпъл­нението на програмата.** |

Време е да видим как се използват статични полета и методи на практика.

**Статични полета и методи – пример**

Примерът, който ще дадем решава следната проста задача: нужен ни е метод, който всеки път връща стойност с едно по-голяма от стойността, върната при предишното извикване на метода. Избираме първата върната от метода стойност да бъде 0. Очевидно такъв метод генерира редицата на естествените числа. Подобна функционалност има широко приложение в практиката, например за унифицирано номериране на обекти. Сега ще видим как може да се реализира с инстру­ментите на ООП.

Да приемем, че методът е наречен **NextValue()** и е дефиниран в клас с име **Sequence**. Класът има поле **currentValue** от тип **int**, което съдържа последно върнатата стойност от метода. Искаме в тялото на метода да се извършват последователно следните две действия: да се увеличава стойността на полето и да се връща като резултат новата му стойност. Връщаната от метода стойност очевидно не зависи от конкретна инстан­ция на класа **Sequence**. Поради тази причина методът и полето са статични. Следва описаната реализация на класа:

|  |
| --- |
| public class Sequence{ // Static field, holding the current sequence value private static int currentValue = 0;  // Intentionally deny instantiation of this class private Sequence() { }  // Static method for taking the next sequence value public static int NextValue() { currentValue++; return currentValue; }} |

Наблюдателният читател е забелязал, че така дефинираният клас има конструктор по подразбиране, който е деклариран като **private**. Тази употреба на конструктор може да изглежда особена, но е съвсем умиш­лена. Добре е да знаем следното:

|  |  |
| --- | --- |
| clip_image007[1] | **Клас, който има само private конструктори не може да бъде инстанциран. Такъв клас обикновено има само ста­тични членове и се нарича utility клас.** |

Засега няма да навлизаме в детайли за употребата на **модификаторите за достъп** **public**, **private** и **protected**. Ще ги разгледаме подробно в главата "[Дефиниране на класове](http://www.introprogramming.info/intro-csharp-book/read-online/glava14-definirane-na-klasove)".

Нека сега видим една проста програма, която използва класа **Sequence**:

|  |
| --- |
| class SequenceManipulating{ static void Main() { Console.WriteLine("Sequence[1..3]: {0}, {1}, {2}", Sequence.NextValue(), Sequence.NextValue(), Sequence.NextValue()); }} |

Примерът извежда на стандартния изход първите три естествени числа чрез трикратно последователно извикване на метода **NextValue()** от класа **Sequence.** Резултатът от този код е следният:

|  |
| --- |
| Sequence[1..3]: 1, 2, 3 |

Ако се опитаме да създадем няколко различни редици, понеже конструкторът на класа **Sequence** е деклариран като **private**, ще получим грешка по време на компилация.

**Примери за системни C# класове**

След като вече се запознахме с основната функционалност на обектите, ще разгледаме накратко няколко често използвани системни класа от стандартните библиотеки на .NET Framework. По този начин ще видим на практика обясненото до момента, а също ще покажем как системните класове улесняват ежедневната ни работа.

**Класът System.Environment**

Започваме с един от основните системни класове в .NET Framework. Той съдържа набор от полезни полета и методи, които улесняват получава­нето на информация за хардуера и операционната система, а някои от тях дават възможност за взаимодействие с обкръжението на програмата. Ето част от функцио­налността, която предоставя този клас:

-     Информация за броя на процесорите, мрежовото име на компютъра, версията на операционната система, името на текущия потребител, текущата директория и др.

-     Достъп до външно дефинирани свойства (properties) и променливи на обкръжението (environment variables), които няма да разглеж­даме в настоящата книга.

Сега ще покажем едно интересно приложение на метод от класа **Environment**, което често се използва в практиката при разработката на програми с критично бързодействие. Ще засечем времето за изпълнение на фрагмент от изходния код с помощта на свойството **TickCount**. Ето как може да стане това:

|  |
| --- |
| class SystemTest{ static void Main() { int sum = 0; int startTime = Environment.TickCount;  // The code fragment to be tested for (int i = 0; i < 10000000; i++) { sum++; }  int endTime = Environment.TickCount; Console.WriteLine("The time elapsed is {0} sec.", (endTime - startTime) / 1000.0); }} |

Статичното свойство **TickCount** от класа **Environment** връща като резултат броя милисекунди, които са изтекли от включването на компютъра до момента на извикването на метода. С негова помощ засичаме изтеклите милисекунди преди и след изпълнението на критичния код. Тяхната разлика е всъщност търсеното време за изпълнение на фрагмента код, измерено в мили­секунди.

В резултат от изпълнението на програмата на стандартния изход се извежда резултат от следния вид (засеченото време варира в зависимост от конкретната компютърна конфигурация и нейното натоварване):

|  |
| --- |
| The time elapsed is 0,031 sec. |

В примера използвахме два статични члена от два системни класа: статичното свойство **Environment.TickCount** и статичния метод **Console. WriteLine(…)**.

**Класът System.String**

Вече сме споменавали класа **string** (**System.String**) от .NET Framework, който представя символни низове (последо­вателности от символи). Да припомним, че можем да считаме низовете за примитивен тип данни в C#, въпреки че работата с тях се различава до известна степен от работата с другите примитивни типове (цели и реални числа, булеви променливи и др.). Ще се спрем по-подробно на тях в темата "[Символни низове](http://www.introprogramming.info/intro-csharp-book/read-online/glava13-simvolni-nizove)".

**Класът System.Math**

Класът **System.Math** съдържа методи за извършването на основни числови операции като повдигане в степен, логаритмуване, коренуване и някои тригонометрични функции. Ще дадем един прост пример, който илюстрира употребата му.

Искаме да съставим програма, която пресмята лицето на триъгълник по дадени дължини на две от страните и ъгъла между тях в градуси. За тази цел имаме нужда от метода **Sin(…)** и константата **PI** на класа **Math**. С помощта на числото  лесно преобразуваме към радиани въведеният в градуси ъгъл. Следва примерна реализация на описаната логика:

|  |
| --- |
| class MathTest{ static void Main() { Console.WriteLine("Length of the first side:"); double a = double.Parse(Console.ReadLine()); Console.WriteLine("Length of the second side:"); double b = double.Parse(Console.ReadLine()); Console.WriteLine("Size of the angle in degrees:"); int angle = int.Parse(Console.ReadLine());;  double angleInRadians = Math.PI \* angle / 180.0; Console.WriteLine("Face of the triangle: {0}", 0.5 \* a \* b \* Math.Sin(angleInRadians)); }} |

Можем лесно да тестваме програмата като проверим дали пресмята пра­вилно лицето на равностранен триъгълник. За допълнително улесне­ние избираме дължина на страната да бъде 2 – тогава лицето му намираме с добре известната формула:



Въвеждаме последователно числата 2, 2, 60 и на стандартния изход се извежда:

|  |
| --- |
| Face of the triangle: 1,73205080756888 |

**Класът System.Math – още примери**

Както вече видяхме, освен математически методи, класът **Math** дефинира и две добре известни в математиката константи: тригонометричната константа ![clip_image009[1]]() и Неперовото число ***e***. Ето още един пример за тях:

|  |
| --- |
| Console.WriteLine(Math.PI);Console.WriteLine(Math.Е); |

При изпълнение на горния код се получава следния резултат:

|  |
| --- |
| 3.1415926535897932.718281828459045 |

**Класът System.Random**

Понякога в програмирането се налага да използваме случайни числа. Например искаме да генерираме 6 случайни числа в интервала между 1 и 49 (не непременно различни). Това можем да направим използвайки класа **System.Random** и неговия метод **Next()**. Преди да използваме класа **Random** трябва да създадем негова инстанция, при което тя се инициализира със слу­чайна стойност (извлечена от текущото системно време в операцион­ната система). След това можем да генерираме случайно число в интервала **[0…n)** чрез извикване на метода **Next(n)**. Забележете, че този метод може да върне нула, но връща винаги случайно число по-малко от зададената стойност **n**. Затова, ако искаме да получим число в интервала **[1…49]**, трябва да използваме израза **Next(49) + 1**. Следва примерен изходен код на прог­рама, която, използвайки класа **Random**, генерира 6 случайни числа в интервала от 1 до 49:

|  |
| --- |
| class RandomNumbersBetween1and49{ static void Main() { Random rand = new Random(); for (int number = 1; number <= 6; number++) { int randomNumber = rand.Next(49) + 1; Console.Write("{0} ", randomNumber); } }} |

Ето как изглежда един възможен изход от работата на програмата:

|  |
| --- |
| 16 49 7 29 1 28 |

**Класът Random – още един пример**

За да ви покажем колко полезен може да е генераторът на случайни числа в .NET Framework, ще си поставим за задача да генерираме случайна парола, която е дълга между 8 и 15 символа, съдържа поне две главни букви, поне две малки букви, поне една цифра и поне три специални знака. За целта ще използваме следния алгоритъм:

1.  Започваме от празна парола. Създаваме генератор на случайни числа.

2.  Генерираме два пъти по една случайна главна буква и я поставяме на случайна позиция в паролата.

3.  Генерираме два пъти по една случайна малка буква и я поставяме на случайна позиция в паролата.

4.  Генерираме една случайна цифра и я поставяме на случайна позиция в паролата.

5.  Генерираме три пъти по един случаен специален символ и го поставяме на случайна позиция в паролата.

6.  До момента паролата трябва да се състои от 8 знака. За да я допъл­ним до най-много 15 символа, можем случаен брой пъти (между 0 и 7) да вмъкнем на случайна позиция в паролата случаен знак (главна буква или малка буква или цифра или специален символ).

Следва имплементация на описания алгоритъм:

|  |
| --- |
| class RandomPasswordGenerator{ private const string CapitalLetters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"; private const string SmallLetters = "abcdefghijklmnopqrstuvwxyz"; private const string Digits = "0123456789"; private const string SpecialChars = "~!@#$%^&\*()\_+=`{}[]\\|':;.,/?<>"; private const string AllChars = CapitalLetters + SmallLetters + Digits + SpecialChars;  private static Random rnd = new Random();  static void Main() { StringBuilder password = new StringBuilder();  // Generate two random capital letters for (int i = 1; i <= 2; i++) { char capitalLetter = GenerateChar(CapitalLetters); InsertAtRandomPosition(password, capitalLetter); }  // Generate two random small letters for (int i = 1; i <= 2; i++) { char smallLetter = GenerateChar(SmallLetters); InsertAtRandomPosition(password, smallLetter); }  // Generate one random digit char digit = GenerateChar(Digits); InsertAtRandomPosition(password, digit);  // Generate 3 special characters for (int i = 1; i <= 3; i++) { char specialChar = GenerateChar(SpecialChars); InsertAtRandomPosition(password, specialChar); }  // Generate few random characters (between 0 and 7) int count = rnd.Next(8); for (int i = 1; i <= count; i++) { char specialChar = GenerateChar(AllChars); InsertAtRandomPosition(password, specialChar); }  Console.WriteLine(password); }  private static void InsertAtRandomPosition( StringBuilder password, char character) { int randomPosition = rnd.Next(password.Length + 1); password.Insert(randomPosition, character); }  private static char GenerateChar(string availableChars) { int randomIndex = rnd.Next(availableChars.Length); char randomChar = availableChars[randomIndex]; return randomChar; }} |

Нека обясним някои неясни моменти в изходния код. Да започнем от дефини­циите на константи:

|  |
| --- |
| private const string CapitalLetters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";private const string SmallLetters = "abcdefghijklmnopqrstuvwxyz";private const string Digits = "0123456789";private const string SpecialChars = "~!@#$%^&\*()\_+=`{}[]\\|':;.,/?<>";private const string AllChars = CapitalLetters + SmallLetters + Digits + SpecialChars;  |

**Константите** в C# представляват неизменими променливи, чиито стойности се задават по време на инициализацията им в изходния код на програмата и след това не могат да бъдат променяни. Те се декларират с модификатора **const**. Използват се за дефиниране на дадено число или низ, което се използва след това многократно в програмата. По този начин се спестяват повторенията на определени стойности в кода и се позволява лесно тези стойности да се променят чрез промяна само на едно място в кода. Например ако в даден момент решим, че символът "**,**" (запетая) не трябва да се ползва при генерирането на пароли, можем да променим само един ред в програмата (съответната константа) и промя­ната ще се отрази навсякъде, където е използвана константата. Констан­тите в C# се изписват в Pascal Case (думите в името са залепени една за друга, като всяка от тях започва с главна буква, а останалите букви са малки).

Нека обясним и как работят останалите части от програмата. В началото като статична член-променлива в класа **RandomPasswordGenerator** се създава генераторът на случайни числа **rnd**. Понеже тази променлива **rnd** е дефинирана в самия клас (не в **Main()** метода), тя е достъпна от целия клас (от всички негови методи) и понеже е обявена за статична, тя е достъпна и от статичните методи. По този начин навсякъде, където програмата има нужда от случайна целочислена стойност, се използва един и същ генератор на случайни числа, който се инициализира при зареждането на класа **RandomPasswordGenerator**.

Методът **GenerateChar()** връща случайно избран символ измежду мно­жество символи, подадени му като параметър. Той работи много просто: избира случайна позиция в множеството символи (между 0 и броя символи минус 1) и връща символът на тази позиция.

Методът **InsertAtRandomPosition()** също не е сложен. Той избира слу­чайна позиция в **StringBuilder** обекта, който му е подаден и вмъква на тази позиция подадения символ. На класа **StringBuilder** ще обърнем специално внимание в главата "[Символни низове](http://www.introprogramming.info/intro-csharp-book/read-online/glava13-simvolni-nizove)".

Ето примерен изход от програмата за генериране на пароли, която разгле­дахме и обяснихме как работи:

|  |
| --- |
| 8p#Rv\*yTl{tN4 |

**Пространства от имена**

**Пространство от имена** (namespace / package) в ООП наричаме контей­нер за група класове, които са обединени от общ признак или се използ­ват в общ контекст. Пространствата от имена спомагат за една по-добра логическа организация на изходния код като създават семантично разде­ление на класо­вете в катего­рии и улесняват употребата им в програмния код. Сега ще се спрем на пространствата в C# и ще видим как можем да ги използваме.

**Какво представляват пространствата от имена в C#?**

**Пространствата от имена (namespaces)** в C# представляват имену­вани групи класове, които са логически свързани, без да има специално изискване как да бъдат разположени във файловата система. Прието е, обаче, името на папката да съвпада с името на пространството и имената на файловете да съвпадат с имената на класовете, които се съхраняват в тях. Трябва да отбележим, че в някои езици за програми­ране компила­цията на изходния код на дадено пространство зависи от разпределението на елементите на пространството в папки и файлове на диска. В Java, например, така описаната файлова организация на пространствата е напълно задължителна (ако не е спазена, възниква грешка при компила­цията). Езикът C# не е толкова стриктен в това отношение.

Нека сега разгледаме механизма за дефиниране на пространства.

**Дефиниране на пространства от имена**

В случай, че искаме да създадем ново пространство или да създадем нов клас, който ще принадлежи на дадено пространство, във Visual Studio това става автоматично чрез командите в контекстното меню на Solution Explorer (при щракане с десния бутон на мишката върху съответната папка). Solution Explorer по подраз­биране се визуализира като страница в дясната част на интегрираната среда. Ще покажем нагледно как можем да добавим нов клас към вече съществуващото пространство **MyNamespace** чрез контекстното меню на Solution Explorer във Visual Studio:



Тъй като проектът ни се нарича **MyConsoleApplication** и добавяме нов клас в неговата подпапка **MyNamespace**, новосъздаденият клас ще бъде в следното пространство:

|  |
| --- |
| namespace MyConsoleApplication.MyNamespace |

Ако сме дефинирали клас в собствен файл и искаме да го добавим към ново или вече съществуващо пространство, не е трудно да го направим ръчно. Достатъчно е да променим именувания блок с ключова дума **namespace**, в който се намира класа:

|  |
| --- |
| namespace <namespace\_name>{ ...} |

При дефиницията използваме ключовата дума **namespace**, последвана от пълното име на пространството. Прието е имената на пространствата в C# да започват с главна буква и да бъдат изписвани в Pascal Case. Например, ако трябва да направим пространство, което съдържа класове за работа със символни низове, желателно е да го именуваме **StringUtils**, а не **string\_utils**.

**Вложени пространства**

Освен класове, пространствата могат да съдържат в себе си и други пространства (вложени пространства, nested namespaces). По този начин съвсем интуитивно се изгражда йерархия от пространства, която позволя­ва още по-прецизно разделение на класовете според тяхната семантика.

При назоваването на пространствата в йерархията се използва символът **.** за разделител (точкова нотация). Например пространството **System** от .NET Framework съдържа в себе си подпространството **Collections** и така пълното название на вложеното пространство **Collections** добива вида **System.Collections**.

**Пълни имена на класовете**

За да разберем напълно смисъла на пространствата, важно е да знаем следното:

|  |  |
| --- | --- |
| clip_image007[2] | **Класовете трябва да имат уникални имена само в рамките на пространството от имена, в което са дефинирани.** |

Извън дадено пространство може да има класове с произволни имена, без значение дали съвпадат с някои от имената на класовете в простран­ството. Това е така, защото класовете в пространството са определени еднозначно от неговия контекст. Време е да видим как се определя синтактично тази еднозначност.

**Пълно име** на клас наричаме собственото име на класа, предшествано от името на пространството, в което този клас е дефиниран. Пълното име на всеки клас е уникално. Отново се използва точковата нотация:

|  |
| --- |
| <namespace\_name>.<class\_name> |

Нека вземем за пример системния клас **CultureInfo**, дефиниран в пространството **System.Globalization** (вече сме го използвали в темата "[Вход и изход от конзолата](http://www.introprogramming.info/intro-csharp-book/read-online/glava4-vhod-i-izhod-ot-konzolata)"). Съгласно дадената дефиниция, пълното име на този клас е **System.Globalization.CultureInfo**.

В .NET Framework понякога има класове от различни пространства със съвпадащи имена, например:

|  |
| --- |
| System.Windows.Forms.ControlSystem.Web.UI.ControlSystem.Windows.Controls.Control |

**Включване на пространство**

При изграждането на приложения в зависимост от предметната област често се налага многократното използване на класове някое простран­ство. За удобство на програмиста има механизъм за **включване** на прост­ранство към текущия файл със сорс код. След като е включено дадено пространство, всички класове дефинирани в него могат свободно да се използват, без да е необходимо използването на техните пълни имена.

Включването на пространство към файл с изходен код се извършва чрез ключо­вата дума **using** по следния начин:

|  |
| --- |
| using <namespace\_name>; |

Ще обърнем внимание на една важна особеност при включването на пространства по описания начин. Всички класове, които се съдържат директно в пространството **<namespace\_name>** са включени и могат да се използват, но трябва да знаем следното:

|  |  |
| --- | --- |
| clip_image007[3] | **Включването на пространства не е рекурсивно, т.е. при включване на пространство не се включват класовете от вложените в него пространства.** |

Например включването на пространството от имена **System.Collections** не включва автоматично класовете, съдър­жащи се в пространството от имена **System.Collections.Generic**. При употребата им трябва да ги назова­ваме с пълните им имена или да включим изрично пространството, в което се намират.

**Включване на пространство – пример**

За да илюстрираме принципа на включването на пространство, ще разгледаме следната програма, въвежда списъци от числа и брои колко от тях са цели и колко от тях са дробни:

|  |
| --- |
| class NamespaceImportTest{ static void Main() { System.Collections.Generic.List<int> ints = new System.Collections.Generic.List<int>(); System.Collections.Generic.List<double> doubles = new System.Collections.Generic.List<double>();  while (true) { int intResult; double doubleResult; Console.WriteLine("Enter an int or a double:"); string input = Console.ReadLine();  if (int.TryParse(input, out intResult)) { ints.Add(intResult); } else if (double.TryParse(input, out doubleResult)) { doubles.Add(doubleResult); } else { break; } }  Console.Write("You entered {0} ints:", ints.Count); foreach (var i in ints) { Console.Write(" " + i); } Console.WriteLine();  Console.Write("You entered {0} doubles:", doubles.Count); foreach (var d in doubles) { Console.Write(" " + d); } Console.WriteLine(); }} |

За целта програмата използва класа **System.Collections.Generic.List** като го назовава с пълното му име.

Нека сега видим как работи горната програма: въвеждаме последователно стойностите **4**, **1.53**, **0.26**, **7**, **2**, **end**. Получаваме следния резултат на стандартния изход:

|  |
| --- |
| You entered 3 ints: 4 7 2You entered 2 doubles: 1.53 0.26 |

Програмата извършва следното: дава на потребителя възможност да въвежда последователно числа, които могат да бъдат цели или реални. Въвеждането продължава до момента, в който бъде въведена стойност, различна от число. След това на стандартния изход се извеждат два реда съответно с целите и с реалните числа.

За реализацията на описаните действия използваме два помощни обекта съответно от тип **System.Collections.Generic.List<int>** и **System. Collections.Generic.List<double>**. Очевидно е, че пълните имена на класо­вете правят кода непрегледен и труден за четене и създават неудобства. Можем лесно да избегнем този ефект като включим прост­ранството **System.Collections.Generic** и използваме директно класовете по име. Следва промене­ният вариант на горната програма:

|  |
| --- |
| using System.Collections.Generic; class NamespaceImportTest{ static void Main() { List<int> ints = new List<int>(); List<double> doubles = new List<double>(); … }} |