Целта на всяка една програма, която създаваме, е да реши даден проблем или да реализира някаква идея. За да измислим решението, ние първо създа­ваме опростен модел на реалността, който не отразява всички факти от нея, а се фокусира само върху тези, които имат значение за намира­нето на решение на нашата задача. След това, използвайки модела, намираме решение (т.е. създаваме алгоритъма) на нашия проблем и това решение го описваме чрез средствата на даден език за програми­ране.

В днешно време най-често използваният тип езици за програмиране са обектно-ориенти­раните. И тъй като обектно-ориентираното програмиране (ООП) е близко до начина на мислене на човека, то ни дава възмож­ността с лекота да описваме модели на заобикалящата ни среда. Една от причините за това е, че ООП ни предоставя средство, за описание на съвкупността от понятия, които описват обектите във всеки модел. Това средство се нарича клас (class). Понятието клас и дефинирането на собствени класове, различни от системните, е вградена възможност на езика C# и целта на настоящата глава е да се запознаем с него.

**Да си припомним: какво са класовете и обектите?**

**Клас** (**class**) в ООП наричаме описание (спецификация) на даден клас обекти от реал­ността. Класът представлява шаблон, който описва видо­вете състояния и поведението на конкретните обекти (екземплярите), които биват създа­вани от този клас (шаблон).

**Обект** (**object**) наричаме екземпляр създаден по дефиницията (описание­то) на даден клас. Когато един обект е създаден по описанието, което един клас дефинира, казваме, че **обектът е от тип "името на този клас"**.

Например, ако имаме клас **Dog**, описващ някакви характеристики на куче от реалния свят, казваме, че обектите, които са създадени по описанието на този клас (например кученцата "Шаро" и "Рекс") са от тип класa **Dog**. Това означение е същото, както когато казваме, че низът "some string" е от класа **String**. Разликата е, че обектът от тип **Dog** е екземпляр от клас, който не е част от библио­теката с класове на .NET Framework, а е дефиниран от самите нас.

**Какво съдържа един клас?**

Всеки клас съдържа дефиниция на това какви данни трябва да се съдържат в един обект, за да се опише състоянието му. Обектът (конкретния екземпляр от този клас) съдържа самите данни. Тези данни дефинират състоянието му.

Освен състоянието, в класа също се описва и поведението на обектите. Поведението се изразява в действията, които могат да бъдат извършвани от обектите. Средството на ООП, чрез което можем да описваме поведе­нието на обектите от даден клас, е декларирането на методи в класа.

**Елементи на класа**

Сега ще изброим основните елементи на един клас, а по-късно ще разгле­да­ме подробно всеки един от тях.

Основните елементи на класовете в C# са следните:

-     **Декларация на класа (class declaration)** – това е редът, на който декларираме името на класа. Например:

|  |
| --- |
| public class Dog |

-     **Тяло на клас** – по подобие на методите, класовете също имат част, която следва декларацията им, оградена с фигурни скоби – "**{**" и "**}**" между които се намира съдържанието на класа. Тя се нарича тяло на класа. Елементите на класа, които се описват в тялото му са изброени в следващите точки.

|  |
| --- |
| public class Dog{ // ... The body of the class comes here ...}  |

-     **Конструктор (constructor)** – това е псевдометод, който се из­пол­зва за създа­ване на нови обекти. Така изглежда един конструктор:

|  |
| --- |
| public Dog(){  // ... Some code ...} |

-     **Полета (fields)** – те са променливи, декларирани в класа (някъде в лите­ратурата се срещат като **член-променливи**). В тях се пазят данни, които отразяват състоянието на обекта и са нужни за работата на методите на класа. Стойността, която се пази в полетата, отразява конкретното състояние на дадения обект, но съществуват и такива полета, наречени **статични**, които са общи за всички обекти.

|  |
| --- |
| // Field definitionprivate string name; |

-     **Свойства (properties)** – така наричаме характеристиките на даден клас. Обикновено стойността на тези характеристики се пази в полета. Подобно на полетата, свойствата могат да бъдат притежа­вани само от конкретен обект или да са споделени между всички обекти от тип даден клас.

|  |
| --- |
| // Property definitionprivate string Name { get; set; } |

-     **Методи (methods)** – от главата "[Методи](http://www.introprogramming.info/intro-csharp-book/read-online/glava9-metodi)", знаем, че методите представляват именувани блокове програмен код. Те извършват някакви действия и чрез тях реализират поведението на обектите от този клас. В методите се изпълняват алгоритмите и се обработват данните на обекта.

Ето как изглежда един клас, който сме дефинирали сами и който прите­жа­ва елементите, които описахме току-що:

|  |
| --- |
| // Class declarationpublic class Dog{ // Opening brace of the class body  // Field declaration private string name;  // Constructor declaration public Dog() { this.name = "Balkan"; }  // Another constructor declaration public Dog(string name) { this.name = name; }  // Property declaration public string Name { get { return name; } set { name = value; } }  // Method declaration public void Bark() { Console.WriteLine("{0} said: Wow-wow!", name); }} // Closing brace of the class body |

За момента няма да обясняваме в по-големи детайли изложения код, тъй като подробна информация ще бъде дадена при обяснението как се декларира всеки един от елементите на класа.

**Използване на класове и обекти**

В главата "[Създаване и използване на обекти](http://www.introprogramming.info/intro-csharp-book/read-online/glava11-sazdavane-i-izpolzvane-na-obekti)" видяхме подробно как се създават нови обекти от даден клас и как могат да се използват. Сега накратко ще си припомним как ставаше това.

**Как да използваме дефиниран от нас клас?**

За да можем да използваме някой клас, първо трябва да създадем обект от него. За целта използваме ключовата дума **new** в комбинация с някой от конструкторите на класа. Това ще създаде обект от дадения клас (тип).

За да можем да манипулираме новосъздадения обект, ще трябва да го присвоим на променлива от типа на неговия клас. По този начин в тази променлива ще бъде запазена връзка (референция) към него.

Чрез променливата, използвайки точкова нотация, можем да извикваме методите, свойствата на обекта, както и да достъпваме поле­тата (член-променливите) и свойствата му.

**Пример – кучешка среща**

Нека вземем примера от [предходната секция на тази глава](http://www.introprogramming.info/intro-csharp-book/read-online/glava14-definirane-na-klasove/#_%D0%95%D0%BB%D0%B5%D0%BC%D0%B5%D0%BD%D1%82%D0%B8_%D0%BD%D0%B0_%D0%BA%D0%BB%D0%B0%D1%81%D0%B0), където де­фи­ни­рахме класа **Dog**, който описва куче, и добавим метод **Main()** към него. В него ще онагледим казаното току-що:

|  |
| --- |
| static void Main(){ string firstDogName = null; Console.WriteLine("Write first dog name: "); firstDogName = Console.ReadLine();  // Using a constructor to create a dog with specified name Dog firstDog = new Dog(firstDogName);  // Using a constructor to create a dog wit a default name Dog secondDog = new Dog();  Console.WriteLine("Write second dog name: "); string secondDogName = Console.ReadLine();  // Using property to set the name of the dog secondDog.Name = secondDogName;  // Creating a dog with a default name Dog thirdDog = new Dog();  Dog[] dogs = new Dog[] { firstDog, secondDog, thirdDog };  foreach (Dog dog in dogs) { dog.Bark(); }} |

Съответно изходът от изпълнението ще бъде следният:

|  |
| --- |
| Write first dog name:*Bobcho*Write second dog name:*Walcho*Bobcho said: Wow-wow!Walcho said: Wow-wow!Balkan said: Wow-wow! |

В примерната програма, с помощта на **Console.ReadLine()**, получаваме имената на обектите от тип куче, които потребителят трябва да въведе от конзолата.

Присвояваме първия въведен низ на променливата **firstDogName**. След това използваме тази променлива при създаването на първия обект от тип **Dog** – **firstDog**, като я подаваме като параметър на конструктора.

Създаваме втория обект от тип **Dog**, без да подаваме низ за името на кучето на конструктора му. След това, чрез **Console.ReadLine()**, въвеж­даме името на второто куче и получената стойност директно подаваме на свойството **Name**. Извикването му става чрез точкова нотация, приложена към променливата, която пази референция към втория създаден обект от тип **Dog – secondDog.Name**.

Когато създаваме третия обект от тип **Dog**, не подаваме име на кучето на конструктора, нито след това модифицираме подразбиращата се стойност "**Balkan**".

След това създаваме масив от тип **Dog**, като го инициализираме с трите обекта, които току-що създадохме.

Накрая, използваме цикъл, за да обходим масива от обекти от тип **Dog**. На всеки елемент от масива, отново използвайки точкова нотация, извикваме метода **Bark()** за съответния обект чрез **dog.Bark()**.

**Природа на обектите**

Нека припомним, че когато в .NET създадем един обект, той се състои от две части – същинска част от обекта, която съдържа неговите данни и се намира в частта от оперативната памет, наречена динамична памет (heap) и референция към този обект, която се намира в друга част от оперативната памет, където се държат локалните променливи и параметрите на мето­дите, наречена стек (stack).

Например, нека имаме клас **Dog**, на който характеристиките му са име (name), порода (kind) и възраст (age). Създаваме променлива **dog** от този клас. Тази променлива се явява референция (указател) към обекта в динамичната памет (heap).

Референцията е променливата, чрез която достъпваме обекта. На схемата по-долу примерната референция, която има връзка към реалния обект в хийпа, е с името **dog**. В нея, за разлика от променливите от примитивен тип, не се съдържа самата стойност (т.е. данните на самия обект), а адреса, на който те се намират в хийпа:



Когато декларираме една променлива от тип някакъв клас, но не искаме тя да е инициализирана с връзка към конкретен обект, тогава трябва да й присвоим стойност **null**. Ключовата дума **null** в езика C# означава, че една променлива не сочи към нито един обект (липса на стойност):



**Съхранение на собствени класове**

В C# единственото ограничение относно съхранението на наши собствени класове е те да са във файлове с разширение **.cs**. В един такъв файл може да има няколко класа, структури и други типове. Въпреки че компилаторът не го изисква, е препоръчително всеки клас да се съхранява в отделен файл, който съответства на името му, т.е. класът **Dog** трябва да е записан във файл с име **Dog.cs**.

**Вътрешна организация на класовете**

Както знаем от темата "[Създаване и използване на обекти](http://www.introprogramming.info/intro-csharp-book/read-online/glava11-sazdavane-i-izpolzvane-na-obekti)", 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.

Ако искаме да включим в кода си пространствата от имена нужни за работата на класовете, декларирани в даден файл или няколко файла, това трябва да стане чрез т.нар. **using** директиви. Те не са задължителни, но ако ги има, трябва да ги поставим на първите редове от файла, преди декларациите на класове или други типове. В следва­щите параграфи ще разберем за какво по-точно служат те.

След включването на използваните пространства от имена, следва декларирането на пространството от имена на класовете във файла. Както вече знаем, не сме задължени да дефинираме класовете си в простран­ство от имена, но е добра практика да го правим, тъй като разпределянето в пространства от имена помага за по-добрата организация на кода и разграничаването на класовете с еднакви имена.

Пространствата от имена съдържат декларации на класове, структу­ри, интерфейси и други типове данни, както и други пространства от имена. Пример за вложени пространства от имена е пространството от имена **System**, което съдържа пространството от имена **Data**. Името на вложеното пространство е **System.Data**.

Пълното име на класа в .NET Framework е името на класа, предшествано от името на пространството от имена, в което той е деклариран: **<namespace\_name>.<class\_name>**. Чрез **using** директивите можем да из­ползваме типовете от дадено пространство от имена, без да уточняваме пълното му име. Например:

|  |
| --- |
| using System;…DateTime date; |

вместо

|  |
| --- |
| System.DateTime date; |

Ето типичната последователност на декларациите, която трябва да след­ваме, когато създаваме собстве­ни **.cs** файлове:

|  |
| --- |
| // Using directives - optionalusing <namespace1>;using <namespace2>; // Namespace definition - optionalnamespace <namespace\_name>{ // Class declaration class <first\_class\_name> { // ... Class body ... }  // Class declaration class <second\_class\_name> { // ... Class body ... }  // ...  // Class declaration class <n-th\_class\_name> { // ... Class body ... }} |

Декларирането на пространство от имена и съответно включването на пространства от имена са вече обяснени в главата "[Създаване и използ­ване на обекти](http://www.introprogramming.info/intro-csharp-book/read-online/glava11-sazdavane-i-izpolzvane-na-obekti)" и затова няма да ги дискутираме отново.

Преди да продължим, да обърнем внимание на първия ред от горната схе­ма. Вместо включвания на пространства от имена той съдържа коментар. Това не е проблем, тъй като по време на компилация, коментарите се "изчистват" от кода и на първи ред от файла остава включване на пространство от имена.

**Кодиране на файловете. Четене на кирилица и Unicode**

Когато създаваме **.cs** файл, в който да дефинираме класовете си, е добре да помислим за кодирането при съхраняването му във файловата система.

Вътрешно в .NET Framework компилираният код се представя в Unicode кодиране и затова няма проблеми, ако във файла използваме символи, които са от азбуки, различни от латинската, например на кирилица:

|  |
| --- |
| using System; public class EncodingTest{ // Тестов коментар static int години = 4;  static void Main() { Console.WriteLine("години: " + години); }} |

Този код ще се компилира и изпълни без проблем, но за да запазим символите четими в редактора на Visual Studio, трябва да осигурим подходящото кодиране на файла.

За да направим това или ако искаме да използваме различно кодиране от Unicode, трябва да асоциираме съответното кодиране с файла. При отва­ряне на файлове това става по следния начин:

1.  От **File** менюто избираме **Open** и след това **File**.

2.  В прозореца **Open File** натискаме стрелката, съседна на бутона **Open** и избираме **Open With**.

3.  От списъка на прозореца **Open With** избираме **Editor** с **encoding support**, например **CSharp Editor with Encoding**.

4.  Натискаме **[OK]**.

5.  В прозореца **Encoding** избираме съответното кодиране от пада­що­то меню **Encoding**.

6.  Натискаме **[OK]**.



За запаметяване на файлове във файловата система в определено кодиране стъпките са следните:

1.  От менюто **File** избираме **Save As**.

2.  В прозореца **Save File As** натискаме стрелката, съседна на бутона **Save** и избираме **Save with Encoding**.

3.  В **Advanced Save Options** избираме желаното кодиране от списъ­ка **Encoding** (за предпочитане е универсалното кодиране UTF-8).

4.  От **Line Endings** избираме желания вид за край на реда.

Въпреки че имаме възможността да използваме символи от други азбуки, в **.cs** файловете, e препоръчително да пишем всички идентификатори и коментари на английски език, за да може кодът ни да е разбираем за по­вече хора по света.

Представете си само, ако ви се наложи да дописвате код, писан от виетна­мец, където имената на променливите и коментарите са на виет­намски език. Не искате да ви се случва, нали? Тогава се замислете как ще се почувства един виетнамец, ако види променливи и коментари на българ­ски език.

**Модификатори и нива на достъп (видимост)**

Нека си припомним, от главата "[Методи](http://www.introprogramming.info/intro-csharp-book/read-online/glava9-metodi)", че модификатор наричаме ключова дума с помощта, на която даваме допълнителна информация на компилатора за кода, за който се отнася модификаторът.

В C# има четири модификатора за достъп. Те са **public**, **private**, **protected** и **internal.** Модификатори за достъп могат да се използват само пред следните елементи на класа: декларация, полета, свойства и методи на класа.

**Модификатори и нива на достъп**

Както обяснихме, в C# има четири модификатора за достъп – **public**, **private**, **protected** и **internal**. С тях ние ограничаваме или позволяваме достъпа (видимостта) до елементите на класа, пред които те са поставени. Нивата на достъп в .NET биват **public**, **protected**, **internal**, **protected internal** и **private**. В тази глава ще се занимаем подробно само с **public**, **private** и **internal**. Повече за **protected** и **protected** **internal** ще научим в главата "[Принципи на обектно-ориенти­раното програмира­не](http://www.introprogramming.info/intro-csharp-book/read-online/glava20-principi-na-obektno-orientiranoto-programirane)".

**Ниво на достъп public**

Използвайки модификатора **public**, ние указваме на компилатора, че елементът, пред който е поставен, може да бъде достъпен от всеки друг клас, независимо дали е от текущия проект, от текущото пространство от имена или извън тях. Нивото на достъп **public** определя липса на ограничения върху видимостта, най-малко рестриктивното от всички нива на достъп в C#.

**Ниво на достъп private**

Нивото на достъп **private**, което налага най-голяма рестрикция на види­мостта на класа и елементите му. Модификаторът **private** служи за индикация, че елементът, за който се отнася, не може да бъде достъпван от никой друг клас (освен от класа, в който е дефиниран), дори този клас да се намира в същото пространство от имена. Това ниво на достъп се използва по подразбиране, т.е. се при­ла­га, когато липсва модификатор за достъп пред съответния елемент на класа.

**Ниво на достъп internal**

Модификаторът **internal** се използва, за да се ограничи достъпът до елемента само от файлове от същото асембли, т.е. същия проект във Visual Studio. Когато във Visual Studio направим няколко проекта, класовете от тях ще се компилират в различни асемблита.

**Асембли (assembly)**

**Асембли (assembly)** е колекция от типове и ресурси, която формира логическа единица функционалност. Всички типове в C# и изобщо в .NET Framework могат да съществуват само в асемблита. При всяка компилация на .NET приложение се създава асембли. То се съхранява като файл с разширение **.exe** или **.dll**.

**Деклариране на класове**

Декларирането на клас следва строго определени правила (синтаксис):

|  |
| --- |
| **[<access\_modifier>] class <class\_name>** |

Когато декларираме клас, задължително трябва да използваме ключовата дума **class**. След нея трябва да стои името на класа **<class\_name>**.

Освен ключовата дума **class** и името на класа, в декларацията на класа могат да бъдат използвани някои модификатори, например разгледаните вече модификатори за достъп.

**Видимост на класа**

Нека имаме два класа – **А** и **В**. Казваме, че класът **А**, има достъп до класа **В**, ако може да прави едно от следните неща:

1.  Създава обект (инстанция) от тип класа **В**.

1.  Достъпва определени методи и член-променливи (полета) в класа **В**, в зависимост от нивото на достъп на съответните методи и полета.

Има и трета операция, която може да бъде извършвана с класове, когато видимостта им позволява, наречена **наследяване на клас**, но на нея ще се спрем по-късно, в главата "[Принципи на обектно-ориентираното програмира­не](http://www.introprogramming.info/intro-csharp-book/read-online/glava20-principi-na-obektno-orientiranoto-programirane)".

Както разбрахме, ниво достъп означава "видимост". Ако класът **А** не може да "види" класа **В**, нивото на достъп на методите и полетата в класа **В** нямат значение.

Нивата на достъп, които един невложен клас може да има, са само **public** и **internal**.

**Ниво на достъп public**

Ако декларираме един клас с модификатор за достъп **public**, ще можем да го достъпваме от всеки един клас и от всичко едно пространство от имена, независимо къде се намират те. Това означава, че всеки друг клас ще може да създава обекти от тип този клас и да има достъп до методите и полетата на класа (стига тези полета да имат подходящо ниво на достъп).

Не трябва да забравяме, че ако искаме да използваме клас с ниво на достъп **public** от друго пространство от имена, различно от текущото, трябва да използваме конструкцията за включване на пространства от имена **using** или всеки път да изписваме пълното име на класа.

**Ниво на достъп internal**

Ако декларираме един клас с модификатор за достъп **internal**, той ще бъде достъпен само от същото асембли. Това означава, че само класовете от същото асембли ще могат да създават обекти от тип този клас и да имат достъп до методите и полетата (с подходящо ниво на достъп) на класа. Това ниво на достъп се подразбира, когато не е използван ни­ка­къв модификатор за достъп при декларацията на класа.

Ако във Visual Studio имаме два проекта в общ solution и искаме от единия проект да използваме клас, дефиниран в другия проект, то реферираният клас трябва задължително да е **public**.

**Ниво на достъп private**

За да сме изчерпателни, трябва да споменем, че като модификатор за достъп до клас, може да се използва модификаторът за видимост **private**, но това е свързано с понятието "вътрешен клас" (nested class), което ще разгледаме в секцията "[Вътрешни класове](http://www.introprogramming.info/intro-csharp-book/read-online/glava14-definirane-na-klasove/#_%D0%92%D1%8A%D1%82%D1%80%D0%B5%D1%88%D0%BD%D0%B8_%D0%BA%D0%BB%D0%B0%D1%81%D0%BE%D0%B2%D0%B5_%28nested)".

**Тяло на класа**

По подобие на методите, след декларацията на класа следва неговото тяло, т.е. частта от класа, в която се съдържа програмния код:

|  |
| --- |
| **[<access\_modifier>] class <class\_name>**{ // ... Class body – the code of the class goes here ... } |

Тялото на класа започва с отваряща фигурна скоба "**{**" и завършва със затваряща – "**}**". Класът винаги трябва да има тяло.

**Правила за именуване на класовете**

По подобие на декларирането на име на метод, за създаването на име на клас съществува следния общоприет стандарт:

1.  Имената на класовете започват с главна буква, а останалите букви са малки. Ако името е съставено от няколко думи, всяка дума започ­ва с главна буква, без да се използват разделители между думите.

2.  За имена на класове обикновено се използват съществителни име­на.

3.  Името на класовете е препоръчително да бъде на английски език.

Ето няколко примера за имена на класове, които са правилно именувани:

|  |
| --- |
| DogAccountCarBufferedReader |

Повече за имената на класовете ще научите в главата "[Качествен програмен код](http://www.introprogramming.info/intro-csharp-book/read-online/glava21-kachestven-programen-kod)".

**Ключовата дума this**

Ключовата дума **this** в C# дава достъп до референцията към текущия обект, когато се използва от метод в даден клас. Това е обектът, чийто метод или конструктор бива извикван. Можем да я разглеждаме като ука­зател (референция), дадена ни априори от създателите на езика, с която да достъпваме елементите (полета, методи, конструктори) на собствения ни клас:

|  |
| --- |
| this.myField; // access a field in the classthis.DoMyMethod(); // access a method in the classthis(3, 4); // access a constructor with two int parameters |

За момента няма да обясняваме изложения код. Разяснения ще дадем по-късно, в местата от секциите на тази глава, посветени на елементите на класа (полета, методи, конструктори) и засягащи ключовата дума **this**.

**Полета**

Както стана дума в началото на главата, когато декларираме клас, описваме обект от реалния свят. За описанието на този обект, се фокусираме само върху характеристиките му, които имат отношение към проблема, който ще решава нашата програма.

Тези характеристики на реалния обект ги интерпретираме в деклараци­ята на класа, като декларираме набор от специален тип променливи, наре­чени полета, в които пазим данните за отделните характеристики. Когато създадем обект по описанието на нашия клас, стойностите на полетата, ще съдържат кон­кретните характеристики, с които даден екзем­пляр от класа (обект) се отличава от всички останали обекти от същия клас.

**Деклариране на полета в даден клас**

До момента сме се сблъсквали само с два типа променливи (вж. главата "[Методи](http://www.introprogramming.info/intro-csharp-book/read-online/glava9-metodi)"), в зависимост от това къде са декларирани:

1.  **Локални променливи** – това са променливите, които са дефини­ра­ни в тялото на някой метод (или блок).

2.  **Параметри** – това са променливите в списъка с параметри, които един метод може да има.

В C# съществува и трети вид променливи, наречени **полета** (**fields**) или **член-променливи на класа** (**instance variables**).

Те се декларират в тялото на класа, но извън тялото на блок, метод или конструктор (какво е конструктор ще разгледаме подробно след малко).

|  |  |
| --- | --- |
| clip_image007 | **Полетата се декларират в тялото на класа, но извън тялото на метод, конструктор или блок.** |

Ето един примерен код, в който се декларират няколко полета:

|  |
| --- |
| class SampleClass{ int age; long distance; string[] names; Dog myDog;} |

Формално, декларацията на полетата става по следния начин:

|  |
| --- |
| **[<modifiers>] <field\_type> <field\_name>;** |

Частта **<field\_type>** определя типа на даденото поле. Той може да бъде както примитивен тип (**byte**, **short**, **char** и т.н.) или масив, така и от тип някакъв клас (например **Dog** или **string**).

Частта **<field\_name>** е името на даденото поле. Както при имената на обикно­вените променливи, когато именуваме една член-променлива, трябва да спазваме правилата за именуване на идентификатори в C# (вж. главата "[Примитивни типове и променливи](http://www.introprogramming.info/intro-csharp-book/read-online/glava2-primitivni-tipove-i-promenlivi)").

Частта **<modifiers>** е понятие, с което сме означили както модифика­то­ри­те за достъп, така и други модификатори. Те не са задължителна част от декларацията на едно поле.

Модификаторите и нивата на достъп, позволени в декларацията на едно поле, са обяснени в секцията "[Видимост на полета и методи](http://www.introprogramming.info/intro-csharp-book/read-online/glava14-definirane-na-klasove/#_%D0%92%D0%B8%D0%B4%D0%B8%D0%BC%D0%BE%D1%81%D1%82_%D0%BD%D0%B0_%D0%BF%D0%BE%D0%BB%D0%B5%D1%82%D0%B0)".

В тази глава, от другите модификато­ри, които не са за достъп, и могат да се използват при декларирането на полета на класа, ще обърнем внимание още на **static**, **const** и **readonly**.

**Област на действие (scope)**

Трябва да знаем, че **областта на действие** (**scope**) на едно поле е от реда, на който е декларирано, до затварящата фигурна скоба на тялото на класа.

**Инициализация по време на деклариране**

Когато декларираме едно поле е възможно едновременно с неговата декларация да му дадем първоначална стойност. Начинът, по който става това, е същият както при инициализацията (даването на стойност) на обикновена локална променлива:

|  |
| --- |
| **[<modifiers>] <field\_type> <field\_name> = <initial\_value>;** |

Разбира се, трябва **<initial\_value>** да бъде от типа на полето или някой съвместим с него тип. Например:

|  |
| --- |
| class SampleClass{ int age = 5; long distance = 234; // The literal 234 is of integer type  string[] names = new string[] { "Pencho", "Marincho" }; Dog myDog = new Dog();  // ... Other code ...} |

**Стойности по подразбиране на полетата**

Всеки път, когато създаваме нов обект от даден клас, се заделя област в динамичната памет за всяко поле от класа. След като бъде заделена, тази памет се инициализира автоматично с подразби­ращи стойности за кон­кретния тип поле (занулява се). Полетата, които на се инициализират изрично при декларацията на полето или в някой от конструкторите, се зануляват.

|  |  |
| --- | --- |
| clip_image007[1] | **При създаване на обект всички негови полета се инициа­лизират с подразбиращите се стойности за типа им, освен ако изрично не бъдат инициализирани.** |

В някои езици (като C и C++) новозаделените обекти не се инициализи­рат автоматично с нулеви стойности и това създава условия за допускане на трудни за откриване грешки. Появява се синдромът "ама това вчера работеше" – непредвидимо поведение, при което програмата понякога работи коректно (когато заделената памет съдържа по случай­ност благоприятни стойности), а понякога не работи (когато заделената памет съдържа неблагоприятни стойности. В C# и въобще в .NET платформата този проблем е решен чрез автоматичното зануляване на полетата.

Стойността по подразбиране за всички типове е 0 или неин еквивалент. За най-често използваните типове подразбиращите се стойности са както следва:

|  |  |
| --- | --- |
| **Тип на поле** | **Стойност по подразбиране** |
| **bool** | **false** |
| **byte** | **0** |
| **char** | **'\0'** |
| **decimal** | **0.0M** |
| **double** | **0.0D** |
| **float** | **0.0F** |
| **int** | **0** |
| референция към обект | **null** |

За по-изчерпателна информация може да погледнете темата "[Примитивни типове и променливи](http://www.introprogramming.info/intro-csharp-book/read-online/glava2-primitivni-tipove-i-promenlivi)", секция "Типове данни", подсекция "Видове", където има пълен списък с всички примитивни типове данни в C# и подразбиращите се стойности за всеки един от тях.

Например, ако създадем клас **Dog** и за него дефинираме полета име (**name**), възраст (**age**), дължина (**length**) и дали кучето е от мъжки пол (**isMale**), без да ги инициализираме по време на декларацията им, те ще бъдат автоматично занулени при създаването на обект от този клас:

|  |
| --- |
| public class Dog{ string name; int age; int length; bool isMale;  static void Main() { Dog dog = new Dog();  Console.WriteLine("Dog's name is: " + dog.name); Console.WriteLine("Dog's age is: " + dog.age); Console.WriteLine("Dog's length is: " + dog.length); Console.WriteLine("Dog is male: " + dog.isMale); }} |

Съответно при стартиране на примера като резултат ще получим:

|  |
| --- |
| Dog's name is:Dog's age is: 0Dog's length is: 0Dog is male: False |

**Автоматична инициализация на локални променливи и полета**

Ако дефинираме дадена локална променлива в един метод, без да я ини­циализираме, и веднага след това се опитаме да я използваме (примерно като отпечатаме стойността й), това ще предизвика грешка при компила­ция, тъй като локалните променливи не се инициализират с подразби­ра­щи се стойности по време на тяхното деклариране.

|  |  |
| --- | --- |
| clip_image007[2] | **За разлика от полетата, локалните променливи, не биват инициализирани с подразбираща се стойност при тяхното деклариране.** |

Нека разгледаме един пример:

|  |
| --- |
| static void Main(){ int notInitializedLocalVariable; Console.WriteLine(notInitializedLocalVariable);} |

Ако се опитаме да компилираме горния код, ще получим следното съобщение за грешка:

|  |
| --- |
| Use of unassigned local variable 'notInitializedLocalVariable' |

**Собствени стойности по подразбиране**

Добър стил на програмиране е обаче, когато декларираме полетата на класа си, изрично да ги инициализираме с някаква подразбираща се стойност, дори ако тя е нула. Въпреки, че C# ще занули всяко едно от полетата, ако ги инициализи­ра­ме изрично, ще направим кода по-ясен и по-лесен за възприемане.

Пример за такова инициализиране може да дадем като модифицираме класът **SampleClass** от предходната секция "[Ини­циа­ли­зация по време на деклари­ране](http://www.introprogramming.info/intro-csharp-book/read-online/glava14-definirane-na-klasove/#_%D0%98%D0%BD%D0%B8%D1%86%D0%B8%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F_%D0%BF%D0%BE_%D0%B2%D1%80%D0%B5%D0%BC%D0%B5)":

|  |
| --- |
| class SampleClass{ int age = 0; long distance = 0; string[] names = null; Dog myDog = null;  // ... Other code ...} |

**Модификатори const и readonly**

Както споменахме в началото на тази секция, в декларацията на едно поле е позволено да се използват модификаторите **const** и **readonly**. Те не са модифи­катори за достъп, а се използват за еднократно инициали­зиране на полета. Полета, декларирани като **const** или **readonly** се наричат **константи**. Използват се когато дадена стойност се повтаря на няколко места в програмата. В такива стойността се изнася като константа и се дефинира само веднъж. Пример за константи от .NET Framework са математическите константи **Math.PI** и **Math.E**, както и константите **String.Empty** и **Int32.MaxValue**.

**Константи, декларирани с const**

Полетата, имащи модификатор **const** в декларацията си, трябва да бъдат инициали­зирани при декларацията си и след това стойността им не може да се променя. Те могат да бъдат достъпвани без да има инстанция на класа, тъй като са споделени между всичко обекти на класа. Нещо повече, при компилация на всички места в кода, където се реферират **const** полета, те се заместват със стойността им, сякаш тя е зададена директно, а не чрез константа. По тази причина **const** полетата се наричат още **compile-time константи**, защото се заместват със стойността им по време на компилация.

**Константи, декларирани с readonly**

Модификаторът **readonly** задава полета, чиято стойността не може да се променя след като веднъж е зададена. Полетата, декларирани с **readonly**, позво­ляват еднократна ини­циали­зация или в момента на декларира­не­то им или в конструкторите на класа. По-късно те не могат да се променят. По тази причина **readonly** полетата се наричат още **run-time константи** – константи, защото стойността им не може да се променя след като се зададе първоначално и run-time, защото стойността им се извлича по време на работа на програмата, както при всички останали полета в класа.

Нека онагледим казаното с пример:

|  |
| --- |
| public class ConstReadonlyModifiersTest{ public const double PI = 3.1415926535897932385; public readonly double size;  public ConstReadonlyModifiersTest(int size) { this.size = size; // Cannot be further modified! }  static void Main() { Console.WriteLine(PI); Console.WriteLine(ConstReadonlyModifiersTest.PI); ConstReadonlyModifiersTest t = new ConstReadonlyModifiersTest(5); Console.WriteLine(t.size);  // This will cause compile-time error Console.WriteLine(ConstReadonlyModifiersTest.size); }} |

**Методи**

В главата "[Методи](http://www.introprogramming.info/intro-csharp-book/read-online/glava9-metodi)" подробно се запознахме с това как да декларираме и използваме метод. В тази сек­ция накратко ще припомним казаното там и ще се фокусираме върху някои допълнителни особености при декларирането и създаването на методи.

**Деклариране на методи в даден клас**

Декларирането на методи, както знаем, става по следния начин:

|  |
| --- |
| // Method definition**[<modifiers>] [<return\_type>] <method\_name>([<parameters\_list>])****{** // ... Method’s body ... **[<return\_statement>];****}** |

Задължителните елементи при декларирането на метода са типът на връ­ща­ната стойност **<return\_type>**, името на метода **<method\_name>** и отваря­щата и затварящата кръгли скоби – "**(**" и "**)**".

Списъкът от параметри **<params\_list>** не е задължителен. Използваме го да подаваме някакви данни на метода, който декларираме, ако той има нужда.

Знаем, че ако типът на връщаната стойност **<return\_type>** е **void**, тогава **<return\_statement>** може да участва само с оператора **return** без аргумент, с цел пре­кра­тя­ване действието на метода. Ако **<return\_type>** е различен от **void**, методът задължително трябва да връща резултат чрез ключовата ду­ма **return** с аргумент, който е от тип **<return\_type>** или съвместим с не­го.

Работата, която методът трябва да свърши, се намира в тялото му, заградена от фигурни скоби – "**{**" и "**}**".

Макар че разгледахме някои от модификаторите за достъп, позволени да се използват при декларирането на един метод, в секцията "[Видимост на полета и методи](http://www.introprogramming.info/intro-csharp-book/read-online/glava14-definirane-na-klasove/#_%D0%92%D0%B8%D0%B4%D0%B8%D0%BC%D0%BE%D1%81%D1%82_%D0%BD%D0%B0_%D0%BF%D0%BE%D0%BB%D0%B5%D1%82%D0%B0)" ще разгледаме по-подробно тази тема.

Ще разгледаме модификатора **static** в [секцията "Статични класове (Static classes) и статични членове на класа (static members) на тази глава](http://www.introprogramming.info/intro-csharp-book/read-online/glava14-definirane-na-klasove/#_%D0%A1%D1%82%D0%B0%D1%82%D0%B8%D1%87%D0%BD%D0%B8_%D0%BA%D0%BB%D0%B0%D1%81%D0%BE%D0%B2%D0%B5_%28static).

**Пример – деклариране на метод**

Нека погледнем декларирането на един метод за намиране сбор на две цели числа:

|  |
| --- |
| int Add(int number1, int number2){ int result = number1 + number2; return result;} |

Името, с което сме го декларирали, е **Add**, а типът на връщаната му стойност е **int**. Списъкът му от параметри се състои от два елемента – променливите **number1** и **number2**. Съответно, връщаме стойността на сбо­ра от двете числа като резултат.

**Достъп до нестатичните данни на класа**

В главата "[Създаване и използване на обекти](http://www.introprogramming.info/intro-csharp-book/read-online/glava11-sazdavane-i-izpolzvane-na-obekti)", разгледахме как чрез опера­то­ра точка, можем да достъпим полетата и да извикаме методите на един клас. Нека припомним как можем да достъпваме полета и да извикваме методи на даден клас, които не са статични, т.е. нямат модификатор **static**, в деклара­цията си.

Например, нека имаме клас **Dog**, с поле за възраст – **age**. За да отпечатаме стойността на това поле, е нужно да създадем обект от клас **Dog** и да достъпим полето на този обект чрез точкова нотация**:**

|  |
| --- |
| public class Dog{ int age = 2;  public static void Main() { Dog dog = new Dog(); Console.WriteLine("Dog's age is: " + dog.age); }} |

Съответно резултатът ще бъде:

|  |
| --- |
| Dog's age is: 2 |

**Достъп до нестатичните полетата на класа от нестатичен метод**

Достъпът до стойността на едно поле може да се осъществява не директно чрез оператора точка (както бе в последния пример **dog.age**), а чрез метод или свойство. Нека в класа **Dog** си създадем метод, който връща стойността на полето **age**:

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

Както виждаме, за да достъпим стойността на полето за възрастта, вътре, от самия клас, използваме ключовата дума **this**. Знаем, че ключовата дума **this** е референция към текущия обект, към който се извиква метода. Следователно, в нашия пример, с "**return this.age**", ние казваме "от те­кущия обект (**this**) вземи (използването на оператора точка) стой­ността на полето **age** и го върни като резултат от метода (чрез ключовата дума **return**)". Тогава, вместо в метода **Main()** да достъпваме стойността на полето **age** на обекта **dog**, ние просто ще извикаме метода **GetAge()**:

|  |
| --- |
| static void Main(){ Dog dog = new Dog(); Console.WriteLine("Dog's age is: " + dog.GetAge());} |

Резултатът след тази промяна ще бъде отново същият.

Формално, декларацията за достъп до поле в рамките на класа, е след­ната:

|  |
| --- |
| **this.<field\_name>** |

Нека подчертаем, че този достъп е възможен само от нестатичен код, т.е. метод или блок, който няма модификатор **static**.

Освен за извличане на стойността на едно поле, можем да използваме ключовата дума **this** и за модифициране на полето.

Например, нека декларираме метод **MakeOlder()**, който извикваме всяка го­ди­на на датата на рож­дения ден на нашия домашен люби­мец и който, увеличава възрастта му с една година:

|  |
| --- |
| public void MakeOlder(){ this.age++;} |

За да проверим дали това, което написахме работи коректно, в края на метода **Main()** добавяме следните два реда:

|  |
| --- |
| // One year later, on the birthday date...dog.MakeOlder();Console.WriteLine("After one year dog's age is: " + dog.age); |

След изпълнението, резултатът е следният:

|  |
| --- |
| Dog's age is: 2After one year dog's age is: 3 |

**Извикване нестатичните методи на класа от нестатичен метод**

По подобие на полетата, които нямат **static** в декларацията си, методите, които също не са статични, могат да бъдат извиквани в тялото на класа чрез ключовата дума **this**. Това става, след като към нея, чрез точкова нотация добавим метода, който ни е необходим заедно с аргументите му (ако има параметри):

|  |
| --- |
| **this.<method\_name>(…)** |

Например, нека създадем метод **PrintAge()**, който отпечатва възрастта на обекта от тип **Dog**, като за целта извиква метода **GetAge()**:

|  |
| --- |
| public void PrintAge(){ int myAge = this.GetAge(); Console.WriteLine("My age is: " + myAge);} |

На първия ред от примера указваме, че искаме да получим възрастта (стойността на поле­то **age**) на текущия обект, използвайки метода **GetAge()** на текущия обект. Това става чрез ключовата дума **this**.

|  |  |
| --- | --- |
| clip_image007[3] | **Достъпването на нестатичните елементи на класа (полета и методи) се осъществява чрез ключовата дума this и оператора за достъп – точка.** |

**Достъп до нестатични данни на класа без използване на this**

Когато достъпваме полетата на класа или извикваме нестатичните му ме­то­ди, е възможно, да го направим без ключовата дума **this**. Тогава двата метода, които декларирахме могат да бъдат записани по следния начин:

|  |
| --- |
| public int GetAge(){ return age; // The same like this.age} public void MakeOlder(){ age++; // The same like this.age++} |

Ключовата дума **this** се използва, за да укаже **изрично**, че трябва да се осъществи достъп до нестатично поле на даден клас или извикваме негов нестатичен метод. Когато това изрично уточнение не е необходимо, може да бъде пропускана и директно да се достъпва елемента на класа.

|  |  |
| --- | --- |
| clip_image007[4] | **Когато не е нужно изрично да се укаже, че се осъщест­вява достъп до елемент на класа, ключовата дума this може да бъде пропусната.** |

Въпреки, че се подразбира, ключовата дума **this** често се използва при достъп до полетата на класа, защото прави кода по-лесен за четене и разбиране, като изрично уточнява, че трябва да се направи достъп до член на класа, а не до локална променлива.

**Припокриване на полета с локални променливи**

От секцията "[Деклариране на полета в даден клас](http://www.introprogramming.info/intro-csharp-book/read-online/glava14-definirane-na-klasove/#_%D0%94%D0%B5%D0%BA%D0%BB%D0%B0%D1%80%D0%B8%D1%80%D0%B0%D0%BD%D0%B5_%D0%BD%D0%B0_%D0%BF%D0%BE%D0%BB%D0%B5%D1%82%D0%B0)" по-горе, знаем, че областта на действие на едно поле е от реда, на който е декларирано полето, до затварящата скоба на тялото на класа. Например:

|  |
| --- |
| public class OverlappingScopeTest{ int myValue = 3;  void PrintMyValue() { Console.WriteLine("My value is: " + myValue); }  static void Main() { OverlappingScopeTest instance = new OverlappingScopeTest(); instance.PrintMyValue(); }} |

Този код ще изведе в конзолата като резултат:

|  |
| --- |
| My value is: 3 |

От друга страна, когато имплементираме тялото на един метод, ни се налага да дефинираме локални променливи, които да използваме по време на изпълнение на метода. Както знаем, областта на действие на тези локални променливи започва от реда, на който са декларирани и продължава до затварящата фигурна скоба на тялото на метода. Например, нека добавим този метод в току-що декларирания клас **OverlappingScopeTest**:

|  |
| --- |
| int CalculateNewValue(int newValue){ int result = myValue + newValue; return result;} |

В този случай, локалната променлива, която използваме, за да изчислим новата стойност, е **result**.

Понякога обаче, може името на някоя локална променлива да съвпадне с името на някое поле. Тогава настъпва колизия.

Нека първо погледнем един пример, преди да обясним за какво става въпрос. Нека модифицираме метода **PrintMyValue()** по следния начин:

|  |
| --- |
| void PrintMyValue(){ int myValue = 5; Console.WriteLine("My value is: " + myValue);} |

Ако декларираме така метода, дали той ще се компилира? А ако се компилира, дали ще се изпълни? Ако се изпълни коя стойност ще бъде отпечатана – тази на полето или тази на локалната променлива?

Така деклариран, след като бъде изпълнен методът **Main()**, резултатът, който ще бъде отпечатан, ще бъде:

|  |
| --- |
| My value is: 5 |

Това е така, тъй като C# позволява да се дефинират локални променливи, чиито имена съвпадат с някое поле на класа. Ако това се случи, казваме, че областта на действие на локалната променлива припокрива областта на действие на полето (**scope overlapping**).

Точно затова областта на действие на локалната променлива **myValue** със стойност **5** препокри областта на действие на полето със същото име. Тогава, при отпечатването на стойността, бе използвана стойността на локалната променлива.

Въпреки това, понякога се налага при колизия на имената да бъде използвано полето, а не локалната променлива със същото име. В този случай, за да извлечем стойността на полето, използваме ключовата дума **this**. За целта достъпваме полето чрез оператора точка, приложен към **this**. По този начин еднозначно указваме, че искаме да използваме стой­ността на полето, не на локалната променлива със същото име.

Нека разгледаме отново нашия пример с извеждането на стойността на полето **myValue**:

|  |
| --- |
| void PrintMyValue(){ int myValue = 5; Console.WriteLine("My value is: " + this.myValue);} |

Този път, резултатът от извикването на метода е:

|  |
| --- |
| My value is: 3 |

**Видимост на полета и методи**

В началото на главата разгледахме общите положения с модификаторите и нивата на достъп на елементите на един клас в C#. По-късно се запознахме подробно с нивата на достъп при декларирането на един клас.

Сега ще разгледаме нивата на видимост на полетата и методите в класа. Тъй като полетата и методите са елементи (членове) на класа и имат едни и същи правила при определяне на нивото им на достъп, ще изложим тези прави­ла едновременно.

За разлика от декларацията на клас, при декларирането на полета и методи на класа, могат да бъдат използвани и четирите нива на достъп – **public**, **protected**, **internal** и **private**. Нивото на видимост **protected** ня­ма да бъде разглеждано в тази глава, тъй като е обвързано с наследяването на класове и е обяснено подробно в главата "[Принципи на обектно-ориентираното програмира­не](http://www.introprogramming.info/intro-csharp-book/read-online/glava20-principi-na-obektno-orientiranoto-programirane)".

Преди да продължим, нека припомним, че ако един клас **A**, не е видим (ня­ма достъп) от друг клас **B**, тогава нито един елемент (поле или метод) на класа **A**, не може да бъде достъ­пен от класа **B**.

|  |  |
| --- | --- |
| clip_image007[5] | **Ако два класа не са видими един за друг, то елементите им (полета и методи) не са видими също, независимо с какви нива на достъп са декларирани самите те.** |

В следващите подсекции, към обясненията, ще разглеждаме примери, в които имаме два класа (**Dog** и **Kid**), които са видими един за друг, т.е. все­ки един от класовете може да създава обекти от тип – другия клас и да до­стъп­ва еле­мен­тите му в зависимост от нивото на достъп, с което са декларирани. Ето как изглежда първия клас **Dog**:

|  |
| --- |
| public class Dog{ private string name = "Sharo";  public string Name { get { return this.name; } }  public void Bark() { Console.WriteLine("wow-wow"); }  public void DoSth() { this.Bark(); }} |

В освен полета и методи се използва и свойство Name, което просто връща полето name. Ще разгледаме свойствата след малко, така че за момента се фокусирайте върху останалото.

Кодът на класа **Kid** има следния вид:

|  |
| --- |
| public class Kid{ public void CallTheDog(Dog dog) { Console.WriteLine("Come, " + dog.Name); }  public void WagTheDog(Dog dog) { dog.Bark(); }} |

В момента, всички елементи (полета и методи) на двата класа са деклари­рани с модификатор за достъп **public**, но при обяснението на различните нива на достъп, ще го променяме съответно. Това, което ще ни интере­сува, е как промяната в нивото на достъп на елемен­тите (полета и методи) на класа **Dog** и ще рефлектира върху достъпа до тези елементи, когато този достъп се извършва от:

-     Самото тяло на класа **Dog**.

-     Тялото на класа **Kid**, съответно вземайки предвид дали **Kid** е в пространството от имена (или асембли), в което се намира класа **Dog** или не.

**Ниво на достъп public**

Когато метод или променлива на класа са декларирани с модификатор за достъп **public**, те могат да бъдат достъпвани от други класове, независи­мо дали другите класове са декларирани в същото пространство от имена, в същото асембли или извън него.

Нека разгледаме двата типа достъп до член на класа, които се срещат в нашите класове **Dog** и **Kid**:

|  |  |
| --- | --- |
| clip_image009 | Достъп до член на класа осъществен в самата деклара­ция на класа. |
| clip_image011 | Достъп до член на класа осъществен, чрез референция към обект, създаден в тялото на друг клас |

Когато членовете на двата класа са **public**, се получава следното:

|  |
| --- |
| **Dog.cs** |
|      clip_image009[1]       clip_image009[2] | class Dog{ public string name = "Sharo";  public string Name { get { return this.name; } }  public void Bark() { Console.WriteLine("wow-wow"); }  public void DoSth() { this.Bark(); }} |

|  |
| --- |
| **Kid.cs** |
|    clip_image011[1]  clip_image011[2] | class Kid{ public void CallTheDog(Dog dog) { Console.WriteLine("Come, " + dog.name); }  public void WagTheDog(Dog dog) { dog.Bark(); }} |

Както виждаме, без проблем осъществяваме, достъп до полето **name** и до метода **Bark()** в класа **Dog** от тялото на самия клас. Независи­мо дали класът **Kid** е в пространството от имена на класа **Dog**, можем от тялото му, да до­стъ­пим полето **name** и съответно да извикаме метода **Bark()** чрез операто­ра точка, прило­жен към референцията **dog** към обект от тип **Dog**.

**Ниво на достъп internal**

Когато член на някой клас бъде деклариран с ниво на достъп **internal**, тогава този елемент на класа може да бъде достъпван от всеки клас в същото асембли (т.е. в същия проект във Visual Studio), но не и от класо­вете извън него (т.е. от друг проект във Visual Studio):

|  |
| --- |
| **Dog.cs** |
|      clip_image009[3]       clip_image009[4] | class Dog{ internal string name = "Sharo";  public string Name { get { return this.name; } }  internal void Bark() { Console.WriteLine("wow-wow"); }  public void DoSth() { this.Bark(); }} |

Съответно, за класа **Kid**, разглеждаме двата случая:

-     Когато е в същото асембли, достъпът до елементите на класа **Dog**, ще бъде позволен, независимо дали двата класа са в едно и също пространство от имена или в различни:

|  |
| --- |
| **Kid.cs** |
|    clip_image011[3]  clip_image011[4] | class Kid{ public void CallTheDog(Dog dog) { Console.WriteLine("Come, " + dog.name); }  public void WagTheDog(Dog dog) { dog.Bark(); }} |

-     Когато класът **Kid** е външен за асемблито, в което е деклариран класът **Dog**, тогава достъпът до полето **name** и метода **Bark()** ще е невъзмо­жен:

|  |
| --- |
| **Kid.cs** |
|    clip_image013  clip_image013[1] | class Kid{clip_image014 public void CallTheDog(Dog dog) { Console.WriteLine("Come, " + dog.name); } clip_image015 public void WagTheDog(Dog dog) { dog.Bark(); }} |

Всъщност достъпът до **internal** членовете на класа **Dog** е невъзможен по две причини: недостатъчна видимост на класа и недостатъчна видимост на членовете му. За да се позволи достъп от друго асембли до класа **Dog**, той, е необходимо той да е деклариран като **public** и едновременно с това въпросните му членове да са декларирани като **public**. Ако или класът или членовете му имат по-ниска видимост, достъпът до тях е невъзможен от други асемблита (други Visual Studio проекти).

Ако се опитаме да компилираме класа **Kid**, когато е външен за асемблито, в което се намира класа **Dog**, ще получим грешки при компилация.

**Ниво на достъп private**

Нивото на достъп, което налага най-много ограничения е **private**. Еле­ментите на класа, които са декларирани с модификатор за достъп **private** (или са декларирани без модификатор за достъп, защото тогава **private** се подразбира), не могат да бъдат достъпвани от никой друг клас, освен от класа, в който са декларирани.

Следователно, ако декларираме полето **name** и метода **Bark()** на класа **Dog**, с модификатори **private**, няма проблем да ги достъпваме вътрешно от самия клас **Dog**, но достъп от други класове не е позволен, дори ако са от същото асембли:

|  |
| --- |
| **Dog.cs** |
|      clip_image009[5]       clip_image009[6] | class Dog{ private string name = "Sharo";  public string Name { get { return this.name; } }  private void Bark() { Console.WriteLine("wow-wow"); }  public void DoSth() { this.Bark(); }} |

|  |
| --- |
| **Kid.cs** |
|    clip_image013[2]  clip_image013[3] | class Kid{clip_image014[1] public void CallTheDog(Dog dog) { Console.WriteLine("Come, " + dog.name); } clip_image015[1] public void WagTheDog(Dog dog) { dog.Bark(); }} |

Трябва да знаем, че когато задаваме модификатор за достъп за дадено поле, той най-често трябва да бъде **private**, тъй като така даваме въз­можно най-висока защита на достъпа до стойността на полето. Съответно, достъпът и модификацията на тази стойност от други класове (ако са необходими) ще се осъществяват единствено чрез свойства или методи. Повече за тази техника ще научим в секцията "Капсулация" на главата "[Принципи на обектно-ориенти­раното програмиране](http://www.introprogramming.info/intro-csharp-book/read-online/glava20-principi-na-obektno-orientiranoto-programirane)".

**Как се определя нивото на достъп на елементите на класа?**

Преди да приключим със секцията за видимостта на елементите на един клас, нека направим един експеримент. Нека в класа **Dog** полето **name** и метода **Bark()** са декларирани с модификатор за достъп **private**. Нека съ­що така, декларираме метод **Main()**, със следното съдържание:

|  |
| --- |
| public class Dog{ private string name = "Sharo";  // ...  private void Bark() { Console.WriteLine("wow-wow"); }  // ...  public static void Main() { Dog myDog = new Dog(); Console.WriteLine("My dog's name is " + myDog.name); myDog.Bark(); }} |

Въпросът, който стои пред нас е, ще се компилира ли класът **Dog**, при положение, че сме декларирали елементите на класа с модификатор за достъп **private**, а в същото време ги извикваме с точкова нотация, приложена към променливата **myDog**, в метода **Main()**?

Стартираме компилацията и тя минава **успешно**. Съответно, резултатът от изпълнението на метода **Main()**, който деклари­рах­ме в класа **Dog** ще бъде следният:

|  |
| --- |
| My dog’s name is Sharo Wow-wow |

Всичко се компилира и работи, тъй като мо­ди­фи­ка­торите за достъп до елементите на класа се прилагат на ниво клас, а не на ниво обекти. Тъй като променливата **myDog** е дефинирана в тялото на класа **Dog** (където е разположен и **Main()** метода на програ­мата), можем да достъпваме елементите му (полета и мето­ди) чрез точкова нотация, независимо че са декларирани с ниво на дос­тъп **private**. Ако обаче се опитаме да направим същото от тялото на класа **Kid**, това няма да е възможно, тъй като достъпът до **private** полетата от външен клас не е разрешено.

**Конструктори**

В обектно-ориентираното програмиране, когато създаваме обект от даден клас, е необходимо да извикаме елемент от класа, наречен конструктор.

**Какво е конструктор?**

Конструктор на даден клас, наричаме псевдометод, който няма тип на връщана стойност, носи името на класа и който се извиква чрез ключо­вата дума **new**. Задачата на конструктора е да инициализира заделената за обекта памет, в която ще се съхраня­ват неговите полетата (тези, които не са **static**).

**Извикване на конструктор**

Единственият начин да извикаме един конструктор в C# е чрез ключовата дума **new**. Тя заделя памет за новия обект (в стека или в хийпа според това дали обектът е стойностен или референтен тип), занулява полетата му, извиква конструктора му (или веригата конструктори, образувана при наследяване) и накрая връща референция към новозаделения обект.

Нека разгледаме един пример, от който ще стане ясно как работи кон­структорът. От главата "[Създаване и използване на обекти](http://www.introprogramming.info/intro-csharp-book/read-online/glava11-sazdavane-i-izpolzvane-na-obekti)", знаем как се създава обект:

|  |
| --- |
| Dog myDog = new Dog(); |

В случая, чрез ключовата дума **new**, извикваме конструктора на класа **Dog**, при което се заделя паметта необходима за новосъздадения обект от тип **Dog**. Когато става дума за класове, те се заделят в динамичната памет (хийпа). Нека проследим как протича този процес стъпка по стъпка. Първо се заделя памет за обекта:



След това се инициализират полетата му (ако има такива) с подразбира­щи­те се стойнос­ти за съответните им типове:



Ако създаването на новия обект е завършило успешно, конструкторът връща референция към него, която се присвоява на променливата **myDog**, от тип класа **Dog**:



**Деклариране на конструктор**

Ако имаме класа **Dog**, ето как би изглеждал неговия най-опростен кон­струк­тор:

|  |
| --- |
| public Dog(){} |

Формално, декларацията на конструктора изглежда по следния начин:

|  |
| --- |
| **[<modifiers>] <class\_name>([<parameters\_list>])** |

Както вече знаем, конструкторите приличат на методи, но нямат тип на връщана стойност (затова ги нарекохме псевдометоди).

**Име на конструктора**

В C# задължително името на всеки конструктор съвпада с името на кла­са, в който го декларираме – **<class\_name>**. В примера по-горе, името на конструктора е същото, каквото е името на класа – **Dog**. Трябва да знаем, че както при методите, името на конструктора винаги е следвано от кръгли скоби – "(" и ")".

В C# **не е позволено**, да се декларира метод, който притежава име, което съвпада с името на класа (следователно и с името на конструк­торите). Ако въпреки всичко бъде деклариран метод с името на класа, това ще доведе до грешка при компилация.

|  |
| --- |
| public class IllegalMethodExample{ // Legal constructor public IllegalMethodExample () { }  // Illegal method private string IllegalMethodExample()  { return "I am illegal method!"; }} |

При опит за компилация на този клас, компилаторът ще изведе следното съобщение за грешка:

|  |
| --- |
| SampleClass: member names cannot be the same as their enclosing type |

**Списък с параметри**

По подобие на методите, ако за създаването на обекта са необходими допълнителни данни, конструкторът ги получава чрез списък от пара­метри – **<parameters\_list>**. В примерния конструктор на класа **Dog** няма нужда от допълнителни данни за създаване на обект от такъв тип и затова няма деклариран списък от параметри. Повече за списъка от параметри ще разгледаме в една от следващите секции – "[Деклариране на конструк­тор с параметри](http://www.introprogramming.info/intro-csharp-book/read-online/glava14-definirane-na-klasove/#_%D0%94%D0%B5%D0%BA%D0%BB%D0%B0%D1%80%D0%B8%D1%80%D0%B0%D0%BD%D0%B5_%D0%BD%D0%B0_%D0%BA%D0%BE%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D0%BE%D1%80)".

Разбира се след декларацията на конструктора, следва неговото тяло, което е като тялото на всеки един метод в C#, но по принцип съдържа предимно инициализационна логика, т.е. задава начални стойности на полетата на класа.

**Модификатори**

Забелязваме, че в декларацията на конструктора, може да се добавят модификатори – **<modifiers>**. За модификаторите, които познаваме и които не са модификатори за достъп, т.е. **const** и **static**, трябва да знаем, че само **const** не е позволен за употреба при декларирането на конструк­тори. По-късно в тази глава, в секцията "[Статични конструктори](http://www.introprogramming.info/intro-csharp-book/read-online/glava14-definirane-na-klasove/#_%D0%A1%D1%82%D0%B0%D1%82%D0%B8%D1%87%D0%BD%D0%B8_%D0%BA%D0%BE%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D0%BE%D1%80%D0%B8)" ще научим повече подробности за конструктори декларирани с модификатор **static**.

**Видимост на конструкторите**

По подобие на полетата и методите на класа, конструкторите, могат да бъдат декларирани с нива на видимост **public**, **protected**, **internal, protected internal** и **private**. Нивата на достъп **protected** и **protected internal** ще бъдат обяснени в главата "[Прин­ципи на обектно-ориентираното програмира­не](http://www.introprogramming.info/intro-csharp-book/read-online/glava20-principi-na-obektno-orientiranoto-programirane)". Остана­лите нива на достъп имат същото значение и поведение като при полетата и методите.

**Инициализация на полета в конструктора**

Както обяснихме по-рано, при създаването на нов обект и извикването на конструктор, се заделя памет за нестатичните полетата на обекта от дадения клас и те се инициализират със стойностите по подразбиране за техния тип (вж. секция "[Извикване на конструктор](http://www.introprogramming.info/intro-csharp-book/read-online/glava14-definirane-na-klasove/#_%D0%98%D0%B7%D0%B2%D0%B8%D0%BA%D0%B2%D0%B0%D0%BD%D0%B5_%D0%BD%D0%B0_%D0%BA%D0%BE%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D0%BE%D1%80)").

Освен това, чрез конструкторите най-често инициализираме полетата на класа, със стойности зададени от нас, а не с подразбиращите се за типа.

Например, в примерите, които разглеждахме до момента, винаги полето **name** на обекта от тип **Dog**, го инициализирахме по време на неговата декларация:

|  |
| --- |
| string name = "Sharo"; |

Вместо да правим това по време на декларацията на полето, по-добър стил на програмиране е да му дадем стойност в конструктора:

|  |
| --- |
| public class Dog{ private string name;  public Dog() { this.name = "Sharo"; }  // ... The rest of the class body ...} |

В някои книги се препоръчва, въпреки че инициализираме полетата в конструктора, изрично да присвояваме подразбира­щи­те се за типа им стойности по време на инициализация, с цел да се подобри четимостта на кода, но това е въпрос на личен избор:

|  |
| --- |
| public class Dog{ private string name = null;  public Dog() { this.name = "Sharo"; }  // ... The rest of the class body ...} |

**Инициализация на полета в конструктора – представяне в паметта**

Нека разгледаме подробно, какво прави конструкторът след като бъде извикан и в тялото му инициализираме полетата на класа. Знаем, че при извикване, той ще задели памет за всяко поле и тази памет ще бъде инициализи­рана със стойността по подразбиране.

Ако полетата са от примитивен тип, тогава след подразбиращите се стойности, ще бъдат присвоени новите, които ние подаваме.

В случая, когато полетата са от референтен тип, например нашето поле **name**, конструкторът ще ги инициализира с **null**. След това ще създаде обекта от съответния тип, в случая низа **"Sharo"** и накрая ще се присвои референция към новия обект в съответното поле, при нас – полето **name**.

Същото ще се получи, ако имаме и други полета, които не са примитивни типове и ги инициализираме в конструктора. Например, нека имаме клас, който описва каишка – **Collar**:

|  |
| --- |
| public class Collar{ private int size;  public Collar() { }} |

Нека съответно нашият клас **Dog**, има поле **collar**, което е от тип **Collar** и което инициализираме в конструктора на класа:

|  |
| --- |
| public class Dog{ private string name; private int age; private double length; private Collar collar;  public Dog() { this.name = "Sharo"; this.age = 3; this.length = 0.5; this.collar = new Collar(); }  public static void Main() { Dog myDog = new Dog(); }} |

Нека проследим стъпките, през които минава конструкторът, след като бъде извикан в **Main()** метода. Както знаем, той ще задели памет в хийпа за всички полета, и ще ги инициализира със съответните им подразби­ра­щи се стойности:



След това, конструкторът ще трябва да се погрижи за създаването на обекта за полето **name** (т.е. ще извика конструктора на класа **string**, който ще свърши работата по създаването на низа):



След това нашия конструктор ще запази референция към новия низ в полето **name**:



След това идва ред на създаването на обекта от тип **Collar**. Нашият конструктор (на класа **Dog**), извиква конструктора на класа **Collar**, който заделя памет за новия обект:



След това я инициализира с подразбиращата се стойност за съответния тип:



След това референцията към новосъздадения обект, която конструкторът на класа **Collar** връща като резултат от изпълнението си, се записва в полето **collar**:



Накрая, референцията към новия обект от тип **Dog** се присвоява на локалната променлива **myDog** в метода **Main()**:



Помним, че локалните променливи винаги се съхраняват в областта от оперативната памет, наречена стек, а обектите – в частта, наречена хийп.

**Последователност на инициализиране на полетата на класа**

За да няма обърквания, нека разясним последователността, в която се инициализират полетата на един клас, независимо от това дали сме им дали стойност по време на декларация и/или сме ги инициализирали в конструктора.

Първо се заделя памет за съответното поле в хийпа и тази памет се ини­циализира със стойността по подразбиране на типа на полето. Напри­мер, нека разгледаме отново нашия клас **Dog**:

|  |
| --- |
| public class Dog{ private string name;  public Dog() { Console.WriteLine( "this.name has value of: \"" + this.name + "\""); // ... No other code here ... } // ... Rest of the class body ...} |

При опит да създадем нов обект от тип нашия клас, в конзолата ще бъде отпечатано съответно:

|  |
| --- |
| this.name has value of: ""  |

Втората стъпка на CLR, след инициализирането на полетата със стой­ността по подразбиране за съответния тип е, ако е зада­дена стойност при декларацията на полето, тя да му се присвои.

Така, ако променим реда от класа **Dog**, на който декларираме полето **name**, то първоначално ще бъде инициализирано със стойност **null** и след това ще му бъде присвоена стойността "**Rex**".

|  |
| --- |
| private string name = "Rex"; |

Съответно, при всяко създаване на обект от нашия клас:

|  |
| --- |
| public static void Main(){ Dog dog = new Dog();} |

Ще бъде извеждано:

|  |
| --- |
| this.name has value of: "Rex" |

Едва след тези две стъпки на инициализация на полетата на класа (ини­циализиране със стойностите по подразбиране и евентуално стойността зададена от програмиста по време на декларация на полето), се извиква конструкторът на класа. Едва тогава, полетата получават стойностите, с които са им дадени в тялото на конструктора.

**Деклариране на конструктор с параметри**

В предната секция, видяхме как можем да дадем стойности на полетата, различни от стойностите по подразбиране. Много често, обаче, по време на декларирането на конструктора не знаем какви стойности ще приемат различните полета. За да се справим с този проблем, по подобие на методите с параметри, нужната информация, която трябва за работата на конструктора, му се подава чрез списъка с параметри. Например:

|  |
| --- |
| public Dog(string dogName, int dogAge, double dogLength){ name = dogName; age = dogAge; length = dogLength; collar = new Collar();} |

Съответно, извикването на конструктор с параметри, става по същия начин както извикването на метод с параметри – нужните стойности ги подаваме в списък, чийто елементи са разделени със запетайки:

|  |
| --- |
| public static void Main(){ Dog myDog = new Dog("Bobi", 2, 0.4); // Passing parameters  Console.WriteLine("My dog " + myDog.name + " is " + myDog.age + " year(s) old. " + " and it has length: " + myDog.length + " m.");} |

Резултатът от изпълнението на този **Main()** метод е следния:

|  |
| --- |
| My dog Bobi is 2 year(s) old. It has length: 0.4 m. |

В C# нямаме ограничение за броя на конструкторите, които можем да създадем. Единственото условие е те да се различават по сигнатурата си (какво е сигнатура обяснихме в главата "[Методи](http://www.introprogramming.info/intro-csharp-book/read-online/glava9-metodi)").

**Област на действие на параметрите на конструктора**

По аналогия на областта на действие на променливите в списъка с пара­метри на един метод, променливите в списъка с параметри на един кон­струк­тор имат област на действие от отварящата скоба на конструк­то­ра до затва­рящата такава, т.е. в цялото тяло на конструктора.

Много често, когато декларираме конструктор с параметри, е възможно да именуваме променливите от списъка му с параметри, със същите имена, като имената на полетата, които ще бъдат инициализирани. Нека за при­мер вземем отново конструктора, който декларирахме в предходната секция:

|  |
| --- |
| public Dog(string name, int age, double length){ name = name; age = age; length = length; collar = new Collar();} |

Нека компилираме и изпълним съответно **Main()** метода, който също из­пол­звахме в предходната секция. Ето какъв е резултатът от изпълне­ние­то му:

|  |
| --- |
| My dog is 0 year(s) old. It has length: 0 m |

Странен резултат, нали? Всъщност се оказва, че не е толкова странен. Обяснението е следното – областта, в която действат про­мен­ливите от списъка с параметри на конструктора, припокрива областта на действие на полетата, които имат същите имена в конструктора. По този на­чин не даваме никаква стойност на полетата, тъй като на практика ние не ги достъпваме. Например, вместо на полето **age**, ние присвояваме стойността на променливата **age** на самата нея:

|  |
| --- |
| age = age; |

Както видяхме в секцията "[Припокриване на полета с локални промен­ливи](http://www.introprogramming.info/intro-csharp-book/read-online/glava14-definirane-na-klasove/#_%D0%9F%D1%80%D0%B8%D0%BF%D0%BE%D0%BA%D1%80%D0%B8%D0%B2%D0%B0%D0%BD%D0%B5_%D0%BD%D0%B0_%D0%BF%D0%BE%D0%BB%D0%B5%D1%82%D0%B0)", за да избегнем това разминаване, трябва да достъ­пим полето, на което искаме да присвоим стойност, но чието име съвпада с името на променлива от списъка с параметри, използвайки ключовата дума **this**:

|  |
| --- |
| public Dog(string name, int age, double length){ this.name = name; this.age = age; this.length = length; this.collar = new Collar();} |

Сега, ако изпълним отново **Main()** метода:

|  |
| --- |
| public static void Main(){ Dog myDog = new Dog("Bobi", 2, 0.4);  Console.WriteLine("My dog " + myDog.name + " is " + myDog.age + " year(s) old. " + " and it has length: " + myDog.length + " m");} |

Резултатът ще бъде точно какъвто очакваме да бъде:

|  |
| --- |
| My dog Bobi is 2 year(s) old. It has length: 0.4 m |

**Конструктор с променлив брой аргументи**

Подобно на методите с променлив брой аргументи, които разгледахме в главата "[Методи](http://www.introprogramming.info/intro-csharp-book/read-online/glava9-metodi)", конструкторите също могат да бъдат декла­ри­ра­ни с параметър за променлив брой аргументи. Правилата за деклара­ция и извикване на конструктори с променлив брой аргументи са същите като тези, които описахме за декларацията и извикването при методи:

-     Когато декларираме конструктор с променлив брой параметри, тряб­ва да използваме запазената дума **params**, след което поставяме типа на параметрите, следван от квадратни скоби. Накрая, следва името на масива, в който ще се съхраняват подадените при извикване на метода аргументи. Например за целочислени аргу­менти ползваме **params int[] numbers**.

-     Позволено е, конструкторът с променлив брой параметри да има и други параметри в списъка си от параметри.

-     Параметърът за променлив брой аргументи трябва да е последен в списъка от параметри на конструктора.

Нека разгледаме примерна декларация на конструктор на клас, който описва лекция:

|  |
| --- |
| public Lecture(string subject, params string[] studentsNames) { // ... Initialization of the instance variables ...} |

Първият параметър в декларацията е името на предмета, по който е лек­цията, а следващия параметър е за променлив брой аргументи – имената на студентите. Ето как би изглеждало примерното създаване на обект от този клас:

|  |
| --- |
| Lecture lecture =  new Lecture("Biology", "Pencho", "Mincho", "Stancho"); |

Съответно, като първи параметър сме подали името на предмета – **"Biology"**, а за всички оставащи аргументи – имената на присъстващите студенти.

**Варианти на конструкторите (overloading)**

Както видяхме, можем да декларираме конструктори с параметри. Това ни дава възможност да декларираме конструктори с различна сигнатура (брой и подредба на параметрите), с цел да предоставим удобство на тези, които ще създават обекти от нашия клас. Създа­ва­нето на конструк­тори с различна сигнатура се нарича създаване на **варианти на кон­структорите (constructors overloading)**.

Нека вземем за пример класа **Dog**. Можем да декларираме различни кон­струк­тори:

|  |
| --- |
| // No parameterspublic Dog(){ this.name = "Sharo"; this.age = 1; this.length = 0.3; this.collar = new Collar();} // One parameterpublic Dog(string name){ this.name = name; this.age = 1; this.length = 0.3; this.collar = new Collar();} // Two parameterspublic Dog(string name, int age){ this.name = name; this.age = age; this.length = 0.3; this.collar = new Collar();} // Three parameterspublic Dog(string name, int age, double length){ this.name = name; this.age = age; this.length = length; this.collar = new Collar();} // Four parameterspublic Dog(string name, int age, double length, Collar collar){ this.name = name; this.age = age; this.length = length; this.collar = collar;} |

**Преизползване на конструкторите**

В последния пример, който дадохме, видяхме, че в зависимост от нуждите за създаване на обекти от нашия клас може да декларираме различни варианти на конструктори. Лесно се забелязва, че голяма част от кода на тези конструктори се повтаря. Това ни кара да се замислим, дали няма начин един конструктор, който вече извършва дадена инициализа­ция, да бъде преизползван от другите, които трябва да изпълнят същата инициа­лизация. От друга страна, в началото на главата споменахме, че един конструктор не може да бъде извикан по начина, по който се извикват методите, а само чрез ключовата дума **new**. Би трябвало да има някакъв начин, иначе много код ще се повтаря излишно.

В C#, съществува механизъм, чрез който един конструктор може да извиква друг конструктор деклариран в същия клас. Това става отново с ключовата дума this, но използвана в друга синтактична конструкция при декларацията на конструкторите:

|  |
| --- |
| [<modifiers>] <class\_name>([<parameters\_list\_1>])  **: this([<parameters\_list\_2>])** |

Към познатата ни форма за деклариране на конструктор (първия ред от декларацията показана по-горе), можем да добавим двоеточие, следвано от ключовата дума **this**, следвана от скоби. Ако конструкторът, който искаме да извикаме е с параметри, в скобите трябва да добавим списък от параметри **parameters\_list\_2**, които да му подадем.

Ето как би изглеждал кодът от предходната секция, в който вместо да повта­ряме инициализацията на всяко едно от полетата, извикваме конструк­тори, декларирани в същия клас:

|  |
| --- |
| // Nо parameterspublic Dog() : this("Sharo") // Constructor call{ // More code could be added here} // One parameterpublic Dog(string name)  : this(name, 1) // Constructor call{} // Two parameterspublic Dog(string name, int age)  : this(name, age, 0.3) // Constructor call{} // Three parameterspublic Dog(string name, int age, double length)  : this(name, age, length, new Collar()) // Constructor call{} // Four parameterspublic Dog(string name, int age, double length, Collar collar){ this.name = name;  this.age = age;  this.length = length;  this.collar = collar;} |

Както е указано чрез коментар в първия конструктор от примера по-горе, ако е необходимо, в допълнение към извикването на някой от другите конструктори с определени параметри всеки конструктор може в тялото си да добави код, който прави допълнителни инициализации или други действия.

**Конструктор по подразбиране**

Нека разгледаме следния въпрос – какво става, ако не декларираме кон­струк­тор в нашия клас? Как ще създа­дем обекти от този тип?

Тъй като често се случва даден клас да няма нито един конструктор, този въпрос е решен в езика C#. Когато не декларираме нито един конструк­тор, компила­то­рът ще създаде един за нас и той ще се използва при създаването на обекти от типа на нашия клас. Този конструктор се нарича **конструктор по подразби­ране** (**default implicit constructor**), който няма да има параметри и ще бъде празен (т.е. няма да прави нищо в допълнение към подразбиращото се зануляване на полетата на обекта).

|  |  |
| --- | --- |
| clip_image007[6] | **Когато не дефинираме нито един конструктор в даден клас, компилаторът ще създаде един, наречен конструк­тор по подразбиране.** |

Например, нека декларираме класа **Collar**, без да декларираме никакъв кон­струк­тор в него:

|  |
| --- |
| public class Collar{ private int size;  public int Size { get { return size; } }} |

Въпреки, че нямаме изрично деклариран конструктор без параметри, ще можем да създадем обекти от този клас по следния начин:

|  |
| --- |
| Collar collar = new Collar(); |

Конструкторът по подразбиране изглежда по следния начин:

|  |
| --- |
| **<access\_level> <class\_name>() { }** |

Трябва да знаем, че конструкторът по подразбиране винаги носи името на класа **<class\_name>** и винаги списъкът му с параметри е празен и неговото тяло е празно. Той просто се "подпъхва" от компилатора, ако в класа няма нито един конструктор. Подразбиращият се конструктор обикновено е **public** (с изключение на някои много специфични ситуации, при които е **protected**).

|  |  |
| --- | --- |
| clip_image007[7] | **Конструкторът по подразбиране е винаги без парамет­ри.** |

За да се уверим, че конструкторът по подразбиране винаги е без пара­мет­ри, нека направим опит да извикаме подразбиращия се конструк­тор, като му подадем параметри:

|  |
| --- |
| Collar collar = new Collar(5); |

Компилаторът ще изведе следното съобщение за грешка:

|  |
| --- |
| 'Collar' does not contain a constructor that takes 1 arguments |

**Работа на конструктора по подразбиране**

Както се досещаме, единственото, което конструкторът по подразбиране ще направи при създаването на обекти от нашия клас, е да занули полетата на класа. Например, ако в класа **Collar** не сме де­кларирали нито един конструктор и създадем обект от него и се опитаме да отпечатаме стойността в полето **size**:

|  |
| --- |
| public static void Main(){ Collar collar = new Collar(); Console.WriteLine("Collar's size is: " + collar.Size);}  |

Резултатът ще бъде:

|  |
| --- |
| Collar's size is: 0 |

Виждаме, че стойността, която е запазена в полето **size** на обекта **collar**, е точно стойността по подразбиране за целочисления тип **int**.

**Кога няма да се създаде конструктор по подразбиране?**

Трябва да знаем, че ако декларираме поне един конструктор в даден клас, тогава компилаторът няма да създаде конструктор по подразбиране.

За да проверим това, нека разгледаме следния пример:

|  |
| --- |
| public Collar(int size)  : this(){ this.size = size;} |

Нека това е единственият конструктор на класа **Collar**. В него се опитва­ме да извикаме конструктор без параметри, надявайки се, че компилато­рът ще е създал конструктор по подразбиране за нас (който знаем, че е без параметри). След като се опитаме да компилираме, ще разберем, че това, което се опитваме да направим, е невъзможно:

|  |
| --- |
| 'Collar' does not contain a constructor that takes 0 arguments |

Ако сме декла­ри­рали дори един единствен конструктор в даден клас, компилаторът няма да създаде конструктор по подразбиране за нас.

|  |  |
| --- | --- |
| clip_image007[8] | **Ако декларираме поне един конструктор в даден клас, компилаторът няма да създаде конструктор по подразби­ране за нас.** |

**Разлика между конструктор по подразбиране и конструктор без параметри**

Преди да приключим със секцията за конструкторите, нека поясним нещо много важно:

|  |  |
| --- | --- |
| clip_image007[9] | **Въпреки че конструкторът по подразбиране и този, без параметри, си приличат по сигнатура, те са напълно раз­лични.** |

Разликата се състои в това, че конструкторът по подразбиране (default implicit constructor) се създава от компилатора, ако не декла­ри­раме нито един конструктор в нашия клас, а **конструкторът без пара­метри (default constructor)** го декларираме ние.

Освен това, както обяснихме по-рано, конструкторът по подразбиране винаги ще има ниво на достъп **protected** или **public**, в зависимост от модификатора на достъп на класа, докато нивото на достъп на конструк­тора без параметри изцяло зависи от нас – ние го определяме.

**Свойства (Properties)**

В света на обектно-ориентираното програмиране съществува елемент на класовете, наречен **свойство (property)**, който е нещо средно между поле и метод и служи за по-добра защита на състоянието в класа. В някои езици за обектно-ориентирано програмиране, като С#, Delphi / Free Pascal, Visual Basic, JavaScript, D, Python и др., свойствата са част от езика, т.е. за тях съществува специален механизъм, чрез който се декларират и използват. Други езици, като например Java, не подържат концепцията за свойства и за целта програмистите, трябва да декларират двойка методи (за четене и модификация на свойството), за да се предостави тази функционалност.

**Свойствата в С# – представяне чрез пример**

Използването на свойства е доказано добра практика и важна част от концепциите на обектно-ориентираното програмиране. Създаването на свойство в програмирането става чрез деклариране на два метода – един за достъп (четене) и един за модификация (записване) на стойността на съответното свойство.

Нека разгледаме един пример. Да си представим, че имаме отново клас **Dog**, който описва куче. Характерно свойство за едно куче е, например, цвета му (color). Достъпът до свойството "цвят" на едно куче и съответната му модифика­ция може да осъществим по следния начин:

|  |
| --- |
| // Getting (reading) a propertystring colorName = dogInstance.Color; // Setting (modifying) a propertydogInstance.Color = "black"; |

**Свойства – капсулация на достъпа до полетата**

Основната цел на свойствата е да осигуряват капсулация на състоянието на класа, в който са декларирани, т.е. да го защитят от попадане в невалидни състояния.

**Капсулация (encapsulation)** наричаме скриването на физическото представяне на данните в един клас, така че, ако в последствие променим това представяне, това да не рефлектира върху останалите класове, които използват този клас.

Чрез синтаксиса на C#, това се реализира като декларираме полета (физи­чес­кото представяне на данните) с възможно най-ограничено ниво на видимост (най-често с модификатор **private)** и декларираме достъпът до тези полета (четене и модифициране) да може да се осъществява единствено чрез специални методи за достъп (accessor methods).

**Капсулация – пример**

За да онагледим какво представлява капсулацията, която предоставят свойствата на един клас, както и какво представляват самите свойства, ще разгледаме един пример.

Нека имаме клас, който представя точка от двумерното пространство със свойства координатите (x, y). Ето как би изглеждал той, ако деклари­раме всяка една от координатите, като поле:

|  |
| --- |
| **Point.cs** |
| using System; class Point{ private double x; private double y;  public Point(int x, int y) { this.x = x; this.y = y; }  public double X { get { return x; } set { x = value; } }  public double Y { get { return y; } set { y = value; } }} |

Полетата на обектите от нашия клас (т.е. координатите на точките) са декларирани като private и не могат да бъдат достъпвани чрез точкова нотация. Ако създадем обект от клас **Point**, можем да модифицираме и четем свойствата (координатите) на точката, единствено чрез свойствата за достъп до тях:

|  |
| --- |
| **PointTest.cs** |
| using System; class PointTest{ static void Main() { Point myPoint = new Point(2, 3);  double myPointXCoordinate = myPoint.X; // Access a property double myPointYCoordinate = myPoint.Y; // Access a property  Console.WriteLine("The X coordinate is: " + myPointXCoordinate); Console.WriteLine("The Y coordinate is: " + myPointYCoordinate); }} |

Резултатът от изпълнението на този **Main()** метод ще бъде следният:

|  |
| --- |
| The X coordinate is: 2The Y coordinate is: 3 |

Ако обаче решим, да променим вътрешното представяне на свойствата на точката, например вместо две полета, ги декларираме като едномерен масив с два елемента, можем да го направим, без това да повлияе по някакъв начин на останалите класове от нашия проект:

|  |
| --- |
| **Point.cs** |
| using System; class Point{ private double[] coordinates;  public Point(int xCoord, int yCoord) { this.coordinates = new double[2];  // Initializing the x coordinate coordinates[0] = xCoord;  // Initializing the y coordinate coordinates[1] = yCoord; }  public double XCoord { get { return coordinates[0]; } set { coordinates[0] = value; } }  public double YCoord { get { return coordinates[1]; } set { coordinates[1] = value; } }} |

Резултатът от изпълнението на **Main()** метода няма да се промени и ре­зул­та­тът ще бъде същият, без да променяме дори символ в кода на класа **PointTest**.

Демонстрираното е добър пример за добра капсулация на данните на един обект предоставена от механизма на свойствата. Чрез тях скриваме вътрешното представяне на информацията, като декларираме свойства / методи за достъп до него и ако в последствие настъпи промяна в представянето, това няма да се отрази на другите класове, които използват нашия клас, тъй като те ползват само свойствата му и не знаят как е представена информацията "зад кулисите".

Разбира се, разгледаният пример демонстрира само една от ползите да се опаковат (обвиват) полетата на класа в свойства. Свойствата позволяват още контрол над данните в класа и могат да проверяват дали присвоява­ните стойности свойства са коректни по някакви критерии. Например ако имаме свойство максимална скорост на клас **Car**, може чрез свойства да наложим изискването стойността й да е в диапазона между 1 и 300 км/ч.

**Още за физическото представяне на свойствата в класа**

Както видяхме по-горе, свойствата могат да имат различно представяне в един клас на физическо ниво. В нашия пример, информацията за свойствата на класа **Point** първоначално беше съхранена в две полета, а след това в едно поле-масив.

Ако обаче решим, вместо да пазим информацията за свойствата на точката в полета, можем да я запазим във файл или в база данни и всеки път, когато се наложи да достъпваме съответното свойство, можем да четем / пишем от файла или базата вместо както в предходните примери да използваме полета на класа. Тъй като свойствата се достъпват чрез специални методи (наречени методи за достъп и модификация или accessor methods), които ще разгле­даме след малко, за класовете, които ще използват нашия клас, това как се съхранява информацията няма да има значение (заради добрата капсула­ция!).

В най-честия случай обаче, информацията за свойствата на класа се пази в поле на класа, което има възможно най-стриктното ниво на видимост **private**.

|  |  |
| --- | --- |
| clip_image007[10] | **Няма значение по какъв начин физически ще бъде пазена информацията за свойствата в един C# клас, но обикнове­но това става чрез поле на класа с максимал­но ограниче­но ниво на достъп (private).** |

**Представяне на свойство без декларация на поле**

Нека разгледаме един пример, в който свойството не се пази нито в поле, нито някъде другаде, а се преизчислява при опит за достъп до него.

Нека имаме клас **Rectangle**, който представя геометричната фигура пра­воъ­гъл­ник. Съответно този клас има две полета – за ширина **width** и дължина **height**. Нека нашия клас има и още едно свойство – лице (**area**). Тъй като винаги чрез дължината и ширината на правоъгълника можем да намерим стойността на свойството "лице", не е нужно да имаме отделно поле в класа, за да пазим тази стойност. По тази причина, можем да си декларираме просто един метод за получаване на лицето, в който пресмятаме формулата за лице на правоъгълник:

|  |
| --- |
| **Rectangle.cs** |
| using System; class Rectangle{ private float height; private float width;  public Rectangle(float height, float width) { this.height = height; this.width = width; }  // Obtaining the value of the property area  public float Area { get { return this.height \* this.width; } }} |

Както ще видим след малко, не е задължително едно свойство да има едновременно методи за модификация и за четене на стойността. Затова е позволено да декларираме само метод за четене на свойството **Area** на правоъгълника. Няма смисъл от метод, който модифицира стойността на лицето на един правоъгълник, тъй като то е винаги едно и също при определена дължина на страните.

**Деклариране на свойства в C#**

За да декларираме едно свойство в C#, трябва да декларираме методи за достъп (за четене и промяна) на съответното свойство и да решим по какъв начин ще съхраняваме ин­фор­ма­цията за това свойство в класа.

Преди да декларираме методите обаче, трябва да декларираме самото свойството в класа. Формално декларацията на свойствата изглежда по следния начин:

|  |
| --- |
| **[<modifiers>] <property\_type> <property\_name>** |

С **<modifiers>** сме означили, както модифика­то­ри­те за достъп, така и други модификатори (например **static**, който ще разгледаме в [следва­щата секция на главата](http://www.introprogramming.info/intro-csharp-book/read-online/glava14-definirane-na-klasove/#_%D0%A1%D1%82%D0%B0%D1%82%D0%B8%D1%87%D0%BD%D0%B8_%D0%BA%D0%BB%D0%B0%D1%81%D0%BE%D0%B2%D0%B5_%28static)). Те не са задължи­тел­­на част от декларацията на едно поле.

Типа на свойството **<property\_type>** задава типа на стойностите на свойството. Може да бъде както примитивен тип (например **int**), така и референтен (например масив).

Съответно, **<property\_name>** е името на свойството. То трябва да започва с главна буква и да удовлетворява правилото **PascalCase**, т.е. всяка нова дума, която се долепя в задната част на името на свойството, започва с главна буква. Ето няколко примера за правилно именувани свойства:

|  |
| --- |
| // MyValue propertypublic int MyValue { get; set; } // Color propertypublic string Color { get; set; } // X-coordinate propertypublic double X { get; set; } |

**Тяло на свойство**

Подобно на класа и методите, свойствата в С# имат тяло, където се декларират методите за достъп до свойството (accessors).

|  |
| --- |
| **[<modifiers>] <property\_type> <property\_name>****{** // ... Property's accessors methods go here**}** |

Тялото на свойството започва с отваряща фигурна скоба "**{**" и завършва със затваряща – "**}**". Свойствата винаги трябва да имат тяло.

**Метод за четене на стойността на свойство (getter)**

Както обяснихме, декларацията на **метод за четене на стойността на едно свойство** (в литературата наричан още **getter**) се прави в тялото на свойството, като за целта трябва да се спазва следния синтаксис:

|  |
| --- |
| **get { <accessor\_body> }** |

Съдържанието на блока ограден от фигурните скоби (**<accessor\_body>**) е подобно на съдържанието на произволен метод. В него се декларират действията, които трябва да се извършат за връщане на резултата от метода.

Методът за четене на стойността на едно свойство трябва да завършва с **return** или **throw** операция. Типът на стойността, която се връща като резултат от този метод, трябва да е същият както типa **<property\_type>** описан в декларацията на свойството.

Въпреки, че по-рано в тази секция срещнахме доста примери на декла­рирани свойства с метод за четене на стойността им, нека разгледаме още един пример за свойството "възраст" (**Age**), което е от тип **int** и е декларирано чрез поле в същия клас:

|  |
| --- |
| private int age; // Field declaration public string Age // Property declaration{ get { return this.age; } // Getter declaration} |

**Извикване на метод за четене на стойността на свойство**

Ако допуснем, че свойството **Age** от последния пример е декларирано в клас от тип **Dog**, извикването на метода за четене на стойността на свойството, става чрез точкова нотация, приложена към променлива от типа, в чийто клас е декларирано свойството:

|  |
| --- |
| Dog dogInstance = new Dog();// ...int dogAge = dogInstance.Age; // Getter invocationConsole.WriteLine(dogInstance.Age); // Getter invocation |

Последните два реда от примера показват, че достъпвайки чрез точкова нотация името на свойството, автоматично се извиква неговият getter метод (методът за четене на стойността му).

**Метод за промяна на стойността на свойство (setter)**

По подобие на метода за четене на стойността на едно свойство, може да се декларира и **метод за промяна (модификация) на стойността на едно свойство** (в литературата наричан още **setter**). Той се декларира в тялото на свойството с тип на връщана стойност **void** и в него подадената при присвояването стойност е достъпна през неявен параметър **value**.

Декларацията се прави в тялото на свойството, като за целта трябва да се спазва следнияt синтаксис:

|  |
| --- |
| **set { <accessor\_body> }** |

Съдържанието на блока ограден от фигурните скоби (**<accessor\_body>**) е подобно на съдържанието, на произволен метод. В него се декларират действията, които трябва да се извършат за промяна на стойността на свойството. Този метод използва неявен параметър, наречен **value**, който е предоставен от С# по подразбиране и който съдържа новата стойност на свойството. Той е от същия тип, от който е свойството.

Нека допълним примера за свойството "възраст" (**Age**) в класа **Dog**, за да онагледим казаното дотук:

|  |
| --- |
| private int age; // Field declaration public string Age // Property declaration{ get{ return this.age; }  set{ this.age = value; } // Setter declaration} |

**Извикване на метод за промяна на стойността на свойство**

Извикването на метода за модификация на стойността на свойството става чрез точкова нотация, приложена към променлива от типа, в чийто клас е декларирано свойството:

|  |
| --- |
| Dog dogInstance = new Dog();// ...dogInstance.Age = 3; // Setter invocation |

На последния ред при присвояването на стойността 3 се извиква setter методът на свойството **Age**, с което тази стойност се записва в параметъра **value** и се подава на setter метода на свойството **Age**. Съответно в нашия пример, стойността на променливата **value** се присвоява на полето **age** oт класа **Dog**, но в общия случай може да се обработи по по-сложен начин.

**Проверка на входните данни на метода за промяна на стойността на свойство**

В процеса на програмиране е добра практика данните, които се подават на setter метода за модификация на свойство да бъдат проверявани дали са валидни и в случай че не са да се вземат необходимите "мерки". Най-често при некоректни данни се предизвиква изключение.

Да вземем отново примера с възрастта на кучето. Както знаем, тя трябва да бъде положително число. За да предотвратим възможността някой да присвои на свойството **Age** стойност, която е отрицателно число или нула, добавяме следната проверка в началото на setter метода:

|  |
| --- |
| public int Age{ get { return this.age; } set { // Take precaution: пerform check for correctness if (value <= 0) { throw new ArgumentException( "Invalid argument: Age should be a negative number."); } // Assign the new correct value this.age = value; }} |

В случай, че някой се опита да присвои стойност на свойството **Age**, която е отрицателно число или 0, ще бъде хвърлено изключение от тип **ArgumentException** с подробна информация какъв е проблемът.

За да се предпази от невалидни данни един клас трябва да проверява подадените му стойности в setter методите на всички свойства и във всички конструктори, както и във всички методи, които могат да променят някое поле на класа. Практиката класовете да се предпазват от нева­лидни данни и невалидни вътрешни състояния се използва широко в програмирането и е част от концепцията "Защитно програмиране", която ще разгледаме в главата "[Качествен програмен код](http://www.introprogramming.info/intro-csharp-book/read-online/glava21-kachestven-programen-kod)".

**Видове свойства**

В зависимост от особеностите им, можем да класифицираме свойствата по следния начин:

1.  **Само за четене (read-only)**, т.е. тези свойства имат само **get** метод, както в примера с лицето на правоъгълник.

2.  **Само за модифициране (write-only)**, т.е. тези свойства имат само **set** метод, но не и метод за четене на стойността на свойството.

3.  И най-честият случай е **read-write**, когато свойството има **методи както за четене, така и за промяна на стойността**.

**Алтернативен похват за работа със свойства**

Преди да приключим секцията ще отбележим още нещо за свойствата в един клас, а именно – как можем да декларираме свойства в С#, без да използваме стандартния синтаксис, разгледан до момента.

В езици за програмиране като Java, в които няма концепция (и съответ­но синтактични средства) за работа със свойства, свойствата се деклари­рат чрез двойка методи, отново наречени getter и setter, по подобие на тези, които разгледахме по-горе.

Тези методи трябва да отговарят на следните изисквания:

1.  **Методът за четене на стойността на свойство** е метод без параметри, който трябва да връща резултат от тип, идентичен с типа на свойството и името му да е образувано от името на свойството с представка **Get**:

|  |
| --- |
| **[<modifiers>] <property\_type> Get<property\_name>** |

2.  Методът за **модификация на стойността на свойство** трябва да има тип на връщаната стойност **void**, името му да е образувано от името на свойството с представка **Set** и типа на единствения аргумент на метода да бъде идентичен с този на свойството:

|  |
| --- |
| **[<modifiers>] void Set<property\_name>(<property\_type> par\_name)** |

Ако представим свойството **Age** на класа **Dog** в примера, който използ­вахме в предходните секции чрез двойка методи, то декларацията на свойството би изглеждала по следния начин:

|  |
| --- |
| private int age; // Field declaration public int GetAge() // Getter declaration{ return this.age;} public void SetAge(int age) // Setter declaration{ this.age = age;} |

Съответно, четенето и модификацията на свойството **Age**, ще се извършва чрез извикване на декларираните методи:

|  |
| --- |
| Dog dogInstance = new Dog(); // ... // Getter invocationsint dogAge = dogInstance.GetAge();Console.WriteLine(dogInstance.GetAge()); // Setter invocationdogInstance.SetAge(3); |

Въпреки че представихме тази алтернатива за декларация на свойства, единствената ни цел бе да бъдем изчерпателни и да направим съпоставка с други езици като Java. Лесно се забелязва, че този начин за декларация на свойствата е по-трудно четим и по-неестествен в сравнение с първия, който изложихме. Затова е препоръчи­телно да се използват вградените средства на езика С# за декларация и използва­не на свойства.

|  |  |
| --- | --- |
| clip_image007[11] | **При работа със свойства е препоръчително да се използва стандартният механизъм, който С# предлага, а не алтер­нативният, който се използва в някои други езици.** |

**Статични класове (static classes) и статични членове на класа (static members)**

Когато един елемент на класа е деклариран с модификатор **static**, го наричаме статичен. В С# като статични могат да бъдат декларирани полетата, методите, свойствата, конструкторите и класовете.

По-долу първо ще разгледаме **статичните елементи** на класа, или с други думи полетата, методите, свойствата и конструкторите му и едва тогава ще се запознаем и с концепцията за статичен клас.

**За какво се използват статичните елементи?**

Преди да разберем принципа, на който работят статичните елементи на класа, нека се запознаем с причините, поради които се налага използ­ването им.

**Метод за сбор на две числа**

Нека си представим, че имаме клас, в който един метод винаги работи по един и същ начин. Например, нека неговата задача е да получи две числа чрез списъка му от параметри и да върне като резултат сбора им. При такъв сценарий няма да има никакво значение кой обект от този клас ще изпълни този метод, тъй като той винаги ще се държи по един и същ начин – ще събира две числа, независими от извикващия обект. Реално **поведението на метода не зависи от състоянието на обекта** (стойностите в полетата на обекта). Тогава защо е нужно да създаваме обект, за да изпълним този метод, при положение че методът не зависи от никой от обектите от този клас? Защо просто не накараме класа да изпълни този метод?

**Брояч на инстанциите от даден клас**

Нека разгледаме и друг сценарий. Да кажем, че искаме да пазим в програмата ни текущия брой на обектите, които са били създадени от даден клас. Как ще съхраним тази променлива, която ще пази броя на създадените обекти?

Както знаем, няма да е възможно да я пазим като поле на класа, тъй като при всяко създаване на обект, ще се създава ново копие на това поле за всеки обект, и то ще бъде инициализирано със стойността по подраз­биране. Всеки обект ще пази свое поле за индикация на броя на обектите и обектите няма да могат да споделят информацията по между си. Изглежда броячът не трябва да е поле в класа, а някак си да бъде извън него. В следващите подсекции ще разберем как да се справим и с този проблем.

**Какво е статичен член?**

Формално погледнато, **статичен член** **(static member)** на класа нари­чаме всяко поле, свойство, метод или друг член, който има модификатор **static** в декларацията си. Това означава, че полета, методи и свойства маркирани като статични, принад­ле­жат на самия клас, а не на някой конкретен обект от да­де­ния клас.

Следователно, когато маркираме поле, метод или свойство като статични, можем да ги използваме, без да създаваме нито един обект от дадения клас. Единственото, от което се нуждаем е да имаме достъп (видимост) до кла­са, за да можем да извикваме статичните му методи, или да достъп­ваме статичните му полета и свойства.

|  |  |
| --- | --- |
| clip_image007[12] | **Статичните елементи на класа могат да се използват без да се създава обект от дадения клас.** |

От друга страна, ако имаме създадени обекти от дадения клас, тогава статичните полета и свойства ще бъдат общи (споделени) за тях и ще има само едно копие на статичното поле или свойство, което се споделя от всички обекти от дадения клас. По тази причина в езика VB.NET вместо ключовата дума **static** със същото значение се ползва ключовата дума **Shared**.

**Статични полета**

Когато създаваме обекти от даден клас, всеки един от тях има различни стойности в полетата си. Например, нека разгледаме отново класа **Dog**:

|  |
| --- |
| public class Dog{ // Instance variables private string name; private int age;} |

Той има две полета съответно за име – **name** и възраст – **age**. Във всеки обект, всяко едно от тези полета има собствена стойност, която се съхранява на различно място в паметта за всеки обект.

Понякога обаче, искаме да имаме полета, които са общи за всички обекти от даден клас. За да постигнем това, трябва в декларацията на тези полета да използваме модификатора **static**. Както вече обяснихме, такива полета се наричат **статични полета (static fields)**. В литерату­рата се срещат, също и като **променливи на класа**.

Казваме, че статичните полета са **асоциирани с класа**, вместо с който и да е обект от този клас. Това означава, че всички обекти, съз­дадени по описанието на един клас **споделят** статичните полета на класа.

|  |  |
| --- | --- |
| clip_image007[13] | **Всички обекти, съз­дадени по описанието на един клас споделят статичните полета на класа.** |

**Декларация на статични полета**

Статичните полета декларираме по същия начин, както се декларира поле на клас, като след модификатора за достъп (ако има такъв), добавяме ключовата дума **static**:

|  |
| --- |
| **[<access\_modifier>] static <field\_type> <field\_name>** |

Ето как би изглеждало едно поле **dogCount**, което пази информация за броя на създадените обекти от клас **Dog**:

|  |
| --- |
| **Dog.cs** |
| public class Dog{ // Static (class) variable static int dogCount;  // Instance variables private string name; private int age;} |

Статичните полета се създават, когато за първи път се опитаме да ги достъпим (прочетем / модифицираме). След създаването си, по подобие на обикновените полета в класа, те се инициализират с подразбиращата се стойност за типа си.

**Инициализация по време на декларация**

Ако по време на декларация на статичното поле, сме задали стойност за инициализация, тя се присвоява на съответното статично поле. Тази инициализация се изпълнява само веднъж – при първото достъпване на полето, веднага след като приключи присвояването на стойността по подразбира­не. При последващо достъпване на полето, тази инициали­зация на статичното поле няма да се изпълни.

В горния примерможем да добавим инициализация на статичното поле:

|  |
| --- |
| // Static variable - declaration and initializationstatic int dogCount = 0; |

Тази инициализация ще се извърши при първото обръщение към статич­ното поле. Когато се опитаме да достъпим някое статично поле на класа, ще се задели памет за него и то ще се инициализира със стойностите по подразбиране. След това, ако полето има инициализация по време на декларацията си (както е в нашия случай с полето **dogCount**), тази иници­ализация ще се извърши. В последствие обаче, когато се опитваме да достъпим полето от други части на програмата ни, този процес няма да се повтори, тъй като статичното поле вече съществува и е инициализирано.

**Достъп до статични полета**

За разлика от обикновените (нестатични) полета на класа, статичните полета, бидейки асоциирани с класа, а не с конкретен обект, могат да бъдат достъпвани от външен клас като към името на класа, чрез точкова нотация, достъпим името на съответното статично поле:

|  |
| --- |
| **<class\_name>.<static\_field\_name>** |

Например, ако искаме да отпечатаме стойността на статичното поле, което пази броя на създадените обекти от нашия клас **Dog**, това ще стане по следния начин:

|  |
| --- |
| public static void Main(){ // Аccess to the static variable through class name Console.WriteLine("Dog count is now " + Dog.dogCount);} |

Съответно, резултатът от изпълнението на този **Main()** метод е:

|  |
| --- |
| Dog count is now 0 |

В C# статичните полета не могат да се достъпват през обект на класа (за разлика от други обектноориентирани езици за програмиране).

Когато даден метод се намира в класа, в който е дефинирано дадено статично поле, то може да бъде достъпено директно без да се задава името на класа, защото то се подразбира:

|  |
| --- |
| **<static\_field\_name>** |

**Модификация на стойностите на статичните полета**

Както вече стана дума по-горе, статичните променливи на класа, са споделени от всички обекти и не принадлежат на нито един обект от класа. Съответно, това дава възможност, всеки един от обектите на класа да променя стойностите на статичните полета, като по този начин останалите обекти ще могат да "видят" модифицираната стойност.

Ето защо, например, за да отчетем броя на създадените обекти от клас **Dog**, е удобно да използваме статично поле, което увеличаваме с единица, при всяко извикване на конструктора на класа, т.е. всеки път, когато създа­ва­ме обект от нашия клас:

|  |
| --- |
| public Dog(string name, int age){ this.name = name; this.age = age;  // Modifying the static counter in the constructor Dog.dogCount += 1; } |

Тъй като осъществяваме достъп до статично поле на класа **Dog** от него самия, можем да си спестим уточняването на името на класа и да ползваме следния код за достъп до полето **dogCount**:

|  |
| --- |
| public Dog(string name, int age){ this.name = name; this.age = age;  // Modifying the static counter in the constructor dogCount += 1;} |

За препоръчване е, разбира се, първият начин, при който е очевидно, че полето е статично в класа **Dog**. При него кодът е по-лесно четим.

Съответно, за да проверим дали това, което написахме е вярно, ще създадем няколко обекта от нашия клас **Dog** и ще отпечатаме броя им. Това ще стане по следния начин:

|  |
| --- |
| public static void Main(){ Dog dog1 = new Dog("Karaman", 1); Dog dog2 = new Dog("Bobi", 2); Dog dog3 = new Dog("Sharo", 3);  // Access to the static variable Console.WriteLine("Dog count is now " + Dog.dogCount);} |

Съответно изходът от изпълнението на примера е:

|  |
| --- |
| Dog count is now 3 |

**Константи (constants)**

Преди да приключим с темата за статичните полета, трябва да се запознаем с един по-особен вид статични полета.

По подобие на константите от математиката, в C#, могат да се създадат специални полета на класа, наречени **константи**. Декларирани и инициализирани веднъж константите, винаги притежават една и съща стойност за всички обекти от даден тип.

В C# константите биват два вида:

1.  Константи, чиято стойност се извлича по време на компилация на програ­мата (compile-time константи).

2.  Константи, чиято стойност се извлича по време на изпълнение на програ­мата (run-time константи).

**Константи инициализирани по време на компилация (compile-time constants)**

Константите, които се изчисляват по време на компилация се декларират по следния начин, използвайки модификатора **const**:

|  |
| --- |
| **[<access\_modifiers>] const <type> <name>;** |

Константите, декларирани със запазената дума **const**, са статични полета. Въпреки това, в декларацията им не се изисква (нито е позволена от компилатора) употребата на модификатора **static**:

|  |  |
| --- | --- |
| clip_image007[14] | **Въпреки, че константите декларирани с модификатор const са статични полета, в декларацията им не трябва и не може да се използва модификаторът static.** |

Например, ако искаме да декларираме като константа числото "пи", познато ни от математиката, това може да стане по следния начин:

|  |
| --- |
| public const double PI = 3.141592653589793; |

Стойността, която присвояваме на дадена константа може да бъде израз, който трябва да бъде изчислим от компилатора по време на компилация. Например, както знаем от математиката, константата "пи" може да бъде представена като приблизителен резултат от делението на числата 22 и 7:

|  |
| --- |
| public const double PI = 22d / 7; |

При опит за отпечатване на стойността на константата:

|  |
| --- |
| public static void Main(){ Console.WriteLine("The value of PI is: " + PI);} |

в командния ред ще бъде изписано:

|  |
| --- |
| The value of PI is: 3.14285714285714 |

Ако не дадем стойност на дадена константа по време на декларацията й, а по-късно, ще получим грешка при компилация. Например, ако в примера с константата **PI**, първо декларираме константата, и по-късно се опитаме да й дадем стойност:

|  |
| --- |
| public const double PI; // ... Some code ... public void SetPiValue(){ // Attempting to initialize the constant PI PI = 3.141592653589793;} |

Компилаторът ще изведе грешка подобна на следната, указвайки ни реда, на който е декларирана константата:

|  |
| --- |
| A const field requires a value to be provided |

Нека обърнем внимание отново:

|  |  |
| --- | --- |
| clip_image007[15] | **Константите декларирани с модификатор const задължи­телно се инициализират в мо­мента на тяхната декларация.** |

**Тип на константите инициализирани по време на компилация**

След като научихме как се декларират константи, които се инициализират по време на компилация, нека разгледаме следния пример: Искаме да създадем клас за цвят (**Color**). Ще използваме т.нар. Red-Green-Blue (RGB) цветови модел, съгласно който всеки цвят е представен чрез смесване на трите основни цвята – червен, зелен и син. Тези три основни цвята са представени като три цели числа в интервала от 0 до 255. Например черното е представено като (0, 0, 0), бялото като (255, 255, 255), синьото – (0, 0, 255) и т.н.

В нашия клас декларираме три целочислени полета за червена, зелена и синя светлина и конструктор, който приема стойности за всяко едно от тях:

|  |
| --- |
| **Color.cs** |
| class Color{ private int red; private int green; private int blue;  public Color(int red, int green, int blue) { this.red = red; this.green = green; this.blue = blue; }} |

Тъй като някои цветове се използват по-често от други (например черно и бяло) можем да декларираме константи за тях, с идеята потребители­те на нашия клас да ги използват наготово вместо всеки път да създават свои собствени обекти за въпросните цветове. За целта модифицираме кода на нашия клас по следния начин, добавяйки декларацията на съответните цветове-константи:

|  |
| --- |
| **Color.cs** |
| class Color{ public const Color Black = new Color(0, 0, 0); public const Color White = new Color(255, 255, 255);  private int red; private int green; private int blue;  public Color(int red, int green, int blue) { this.red = red; this.green = green; this.blue = blue; }} |

Странно, но при опит за компилация, получаваме следната грешка:

|  |
| --- |
| 'Color.Black' is of type 'Color'. A const field of a reference type other than string can only be initialized with null.'Color.White' is of type 'Color'. A const field of a reference type other than string can only be initialized with null. |

Това е така, тъй като в С#, константи, декларирани с модификатор **const**, могат да бъдат само от следните типове:

1.  Примитивни типове: **sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**, **char**, **float**, **double**, **decimal**, **bool**.

2.  Изброени типове (разгледани в секция "[Изброени типове (enumerations)](http://www.introprogramming.info/intro-csharp-book/read-online/glava14-definirane-na-klasove/#_%D0%98%D0%B7%D0%B1%D1%80%D0%BE%D0%B5%D0%BD%D0%B8_%D1%82%D0%B8%D0%BF%D0%BE%D0%B2%D0%B5_%28enumerations%29)" в края на тази глава).

3.  Референтни типове (най-вече типът **string**).

Проблемът при компилацията на класа в нашия пример е свързан с референтните типове и с ограничението на компилатора да не позволява едновременната употреба на оператора **new** при деклариране на константа, когато тази константа е декларирана с модификатора **const**, освен ако референтният тип не може да се изчисли по време на компилация.

Както се досещаме, единственият референтен тип, който може да бъде изчислен по време на компилация при употребата на оператора **new** е **string**.

Следователно, единствените възможности за константи от референтен тип, които са декларирани с модификатор **const**, са следните:

1.  Константите трябва да са от тип **string**.

2.  Стойността, която присвояваме на константата от референтен тип, различен от **string**, е **null**.

Можем да формулираме следната дефиниция:

|  |  |
| --- | --- |
| clip_image007[16] | **Константите декларирани с модификатор const трябва да са от примитивен, изброен или референтен тип като ако са от референтен тип, то този тип трябва да е или string или стойността, която се присвоява на константата трябва да бъде null.** |

Следователно, използвайки модификатора **const** няма да успеем да декларираме константите **Black** и **White** от тип **Color** в нашия клас за цвят, тъй като те не са **null**. Как да решим този проблем, ще видим в следващата подсек­ция.

**Константи инициализирани по време на изпълнение на програмата**

Когато искаме да декларираме константи от референтен тип, които не могат да бъдат изчислени по време на компилация на програмата, вместо модификатора **const**, в декларацията на константата трябва да използ­ваме комбинацията от модификато­ри **static** **readonly**:

|  |
| --- |
| **[<access\_modifiers>] static readonly <reference-type> <name>;** |

Съответно **<reference-type>** е такъв тип, чиято стойност не може да бъде изчислена по време на компилация.

Сега, ако заменим **const** със **static readonly** в последния пример от предходната секция, компилацията минава успешно:

|  |
| --- |
| public static readonly Color Black = new Color(0, 0, 0);public static readonly Color White = new Color(255, 255, 255); |

**Именуване на константите**

Съгласно конвенцията на Microsoft имената на константите в C# следват правилото **PascalCase**. Ако константата е съставена от няколко думи, всяка нова дума след първата започва с главна буква. Ето няколко примера за правилно именувани константи:

|  |
| --- |
| // The base of the natural logarithms (approximate value)public const double E = 2.718281828459045;public const double PI = 3.141592653589793;public const char PathSeparator = '/';public const string BigCoffee = "big coffee";public const int MaxValue = 2147483647;public static readonly Color DeepSkyBlue = new Color(0,104,139); |

Понякога за константите се ползва и именуване в стил ALL-CAPS, но то не се подкрепя официално от код конвенциите на Майкрософт, макар и да е силно разпространено в програмирането:

|  |
| --- |
| public const double FONT\_SIZE\_IN\_POINTS = 14; // 14pt font size |

Както стана ясно от примерите, разликата между **const** и **static readonly** полетата е в момента, в който им се присвояват стойностите. Compile-time константите (**const**) трябва да бъдат инициализирани в момента на декларацията си, докато run-time константите (**static readonly**) могат да бъдат инициализирани на по-късен етап, например в някой от конструк­торите на класа, в който са дефинирани.

**Употреба на константите**

Константите в програмирането се използват, за да се избегне повто­рението на числа, символни низове или други често срещани стойности (литерали) в програмата и да се позволи тези стойности лесно да се променят. Използването на константи вместо твърдо забити в кода повтарящи се стойности улеснява четимостта и поддръжката на кода и е препоръчителна практика. Според някои автори всички литерали, различни от **0**, **1**, **-1**, празен низ, **true**, **false** и **null** трябва да се декларират като константи, но понякога това затруднява четенето и поддръжката на кода вместо да го опрости. По тази причина се счита, че като константи трябва да се обявят стойностите, които се срещат повече от веднъж в програмата или има вероятност да бъдат променени с течение на времето.

Кога и как да използваме ефективно константите ще научим в подроб­ности в главата "[Качествен програмен код](http://www.introprogramming.info/intro-csharp-book/read-online/glava21-kachestven-programen-kod)".

**Статични методи**

По подобие на статичните полета, когато искаме един метод да е асоцииран само с класа, но не и с конкретен обект от класа, тогава го декларираме като статичен.

**Декларация на статични методи**

Синтактично да декларираме **статичен метод** означава, в деклара­цията на метода, да добавим ключовата дума **static**:

|  |
| --- |
| **[<access\_modifier>] static <return\_type> <method\_name>()** |

Нека например декларираме метода за събиране на две числа, за който говорихме в началото на настоящата секция:

|  |
| --- |
| public static int Add(int number1, int number2){ return (number1 + number2);} |

**Достъп до статични методи**

Както и при статичните полета, статичните методи могат да бъдат достъп­вани чрез точкова нотация (операторът точка) приложена към името на класа, като името на класа може да се пропусне ако извикването се извършва от същия клас, в който е деклариран статичният метод. Ето един пример за извикване на статичния метод **Add(…)**:

|  |
| --- |
| public static void Main(){ // Call the static method through its class int sum = MyMathClass.Add(3, 5);  Console.WriteLine(sum);} |

**Достъп между статични и нестатични членове**

В повечето случаи статичните методи се използват за достъпване на статични полета от класа, в който са дефинирани. Например, когато искаме да декларираме метод, който да връща броя на създадените обекти от класа **Dog**, той трябва да бъде статичен, защото нашият брояч също е статичен:

|  |
| --- |
| public static int GetDogCount(){ return dogCount;} |

Но когато разглеждаме как статични и нестатични методи и полета могат да се достъпват, не всички комбинации са позволени.

**Достъп до нестатичните членове на класа от нестатичен метод**

Нестатичните методи могат да достъпват нестатичните полета и други нестатични методи на класа. Например, в класа **Dog** можем да деклари­раме метод **PrintInfo()**, който извежда информация за нашето куче:

|  |
| --- |
| **Dog.cs** |
| public class Dog{ // Static variable static int dogCount;  // Instance variables private string name; private int age;  public Dog(string name, int age) { this.name = name; this.age = age;  dogCount += 1; }  public void Bark() { Console.Write("wow-wow"); }  // Non-static (instance) method public void PrintInfo() { // Accessing instance variables – name and age Console.Write("Dog's name: " + this.name + "; age: " + this.age + "; often says: ");  // Calling instance method this.Bark(); }} |

Разбира се, ако създадем обект от класа **Dog** и извикаме неговия **PrintInfo()** метод:

|  |
| --- |
| public static void Main(){ Dog dog = new Dog("Sharo", 1); dog.PrintInfo();} |

Резултатът ще бъде следният:

|  |
| --- |
| Dog's name: Sharo; age: 1; often says: wow-wow |

**Достъп до статичните елементи на класа от нестатичен метод**

От нестатичен метод, можем да достъпваме статични полета и статични методи на класа. Както разбрахме по-рано, това е така, тъй като статич­ните методи и променливи са обвързани с класа, вместо с конкретен метод и статичните елементи могат да се достъпват от кой да е обект на класа, дори от външни класове (стига да са видими за тях). Например:

|  |
| --- |
| **Circle.cs** |
| public class Circle{ public static double PI = 3.141592653589793;  private double radius;  public Circle(double radius) { this.radius = radius; }  public static double CalculateSurface(double radius) { return (PI \* radius \* radius); }  public void PrintSurface() {  double surface = CalculateSurface(radius); Console.WriteLine("Circle's surface is: " + surface); }} |

В примера от нестатичния метод **PrintSurface()** осъществяваме достъп до стойността на статичното поле **PI**, както извикваме статичния метод **CalculateSurface()**. Нека опитаме да извикаме въпросния нестатичен метод:

|  |
| --- |
| public static void Main(){ Circle circle = new Circle(3); circle.PrintSurface();} |

След компилация и изпълнение, на конзолата ще бъде изве­дено:

|  |
| --- |
| Circle's surface is: 28.2743338823081 |

**Достъп до статичните елементи на класа от статичен метод**

От статичен метод можем да извикваме друг статичен метод или статично поле на класа безпроблемно.

Например, нека вземем нашия клас за математически пресмятания. В него имаме декларирана константата **PI**. Можем да декларираме статичен метод за намиране дължината на окръжност (фор­мулата за намиране периметър на окръжност е **2**π**r**, където **r** е радиусът на окръжността), който за пресмятането на периметъра на дадена окръж­ност, ползва константата **PI**. След това, за да покажем, че статичен метод може да вика друг статичен метод, можем от статичния метод **Мain()** да извикаме статичния метод за намиране периметъра на окръжност:

|  |
| --- |
| **MyMathClass.cs** |
| public class MyMathClass{ public const double PI = 3.141592653589793;  // The method applies the formula: P = 2 \* PI \* r public static double CalculateCirclePerimeter(double r) { // Accessing the static variable PI from static method return (2 \* PI \* r); }  public static void Main() { double radius = 5;  // Accessing static method from other static method double circlePerimeter = CalculateCirclePerimeter(radius);  Console.WriteLine("Circle with radius " + radius + " has perimeter: " + circlePerimeter); }} |

Кодът се компилира без грешки и при изпълнение извежда следния резул­тат:

|  |
| --- |
| Circle with radius 5.0 has perimeter: 31.4159265358979 |

**Достъп до нестатичните елементи на класа от статичен метод**

Нека разгледаме най-интересния случай от комбинацията от достъпване на статични и неста­тич­ни елементи на класа – достъпването на нестатич­ни елементи от статичен метод.

Трябва да знаем, че от статичен метод не могат да бъдат достъпвани нестатични полета, нито да бъдат извиквани нестатични методи. Това е така, защото статичните методи са обвързани с класа, и не "знаят" за нито един обект от класа. Затова, ключовата дума **this** не може да се използва в статични методи – тя е обвързана с конкретна инстанция на класа. При опит за достъпване на нестатични елементи на класа (полета или методи) от статичен метод, винаги ще получаваме грешка при компилация.

**Непозволен достъп до нестатично поле от статичен метод – пример**

Ако в нашия клас **Dog** се опитаме да декларираме статичен метод **PrintName()**, който връща като резултат стойността на нестатичното поле **name** декларирано в класа:

|  |
| --- |
| public void string PrintName(){ // Trying to access non-static variable from static method Console.WriteLine(name); // INVALID} |

Съответно компилаторът ще ни отговори със съобщение за грешка, подобно на следното:

|  |
| --- |
| An object reference is required for the non-static field, method, or property 'Dog.name' |

Ако въпреки това, се опитаме в метода да достъпим полето чрез ключовата дума **this**:

|  |
| --- |
| public void string PrintName(){ // Trying to access non-static variable from static method Console.WriteLine(this.name); // INVALID} |

Компилаторът отново няма да е доволен и този път ще изведе следното съобщение, без да успее да компилира класа:

|  |
| --- |
| Keyword 'this' is not valid in a static property, static method, or static field initializer |

**Непозволено извикване на нестатичен метод от статичен метод – пример**

Сега ще се опитаме да извикаме нестатичен метод от статичен метод. Нека в нашия клас **Dog** декларираме нестатичен метод **PrintAge()**, който отпечатва стойността на полето **age**:

|  |
| --- |
| public void PrintAge(){ Console.WriteLine(this.age);} |

Съответно, нека се опитаме от метода **Main()**, който декларираме в класа **Dog**, да извикаме този метод без да създаваме обект от нашия клас:

|  |
| --- |
| public static void Main(){ // Attempt to invoke non-static method from a static context PrintAge(); // INVALID} |

При опит за компилация ще получим следната грешка:

|  |
| --- |
| An object reference is required for the non-static field, method, or property 'Dog.PrintAge()' |

Резултатът е подобен, ако се опитаме да измамим компилатора, опитвайки се да извикаме метода чрез ключовата дума **this**:

|  |
| --- |
| public static void Main(){ // Attempt to invoke non-static method from a static context this.PrintAge(); // INVALID} |

Съответно, както в случая с опита за достъп до нестатично поле от статичен метод, чрез ключовата дума **this**, компилаторът извежда следното съобщение, без да успее да компилира нашия клас:

|  |
| --- |
| Keyword 'this' is not valid in a static property, static method, or static field initializer |

От разгледаните примери, можем да направим следния извод:

|  |  |
| --- | --- |
| clip_image007[17] | **Нестатичните елементи на класа НЕ могат да бъдат използ­вани в статичен контекст.** |

Проблемът с достъпа до нестатични елементи на класа от статичен метод има едно единствено решение – тези нестатични елементи да се достъпват чрез референция към даден обект:

|  |
| --- |
| public static void Main(){ Dog myDog = new Dog("Sharo", 2); string myDogName = myDog.name; Console.WriteLine("My dog \"" + myDogName + "\" has age of "); myDog.PrintAge(); Console.WriteLine("years");} |

Съответно този код се компилира и резултатът от изпълнението му е:

|  |
| --- |
| My dog "Sharo" has age of 2 years |

**Статични свойства на класа**

Макар и рядко, понякога е удобно да се декларират и използват свойства не на обекта, а на класа. Те носят същите характеристики като свой­ствата, свързани с конкретен обект от даден клас, които разгледахме по-горе, но с тази разлика, че статичните свойства се отнасят за класа.

Както можем да се досетим, всичко, което е нужно да направим, за да превърнем едно обикновено свойство в статично, е да добавим ключовата ма **static** при декларацията му.

Статичните свойства се декларират по следния начин:

|  |
| --- |
| **[<modifiers>] static <property\_type> <property\_name>****{** // ... Property’s accessors methods go here**}** |

Нека разгледаме един пример. Имаме клас, който описва някаква система. Можем да създаваме много обекти от нея, но моделът на системата има дадена версия и производител, които са общи за всички екземпляри, създадени от този клас. Можем да направим версията и производи­телите статични свойства на класа:

|  |
| --- |
| **SystemInfo.cs** |
| public class SystemInfo{ private static double version = 0.1; private static string vendor = "Microsoft";  // The "version" static property public static double Version { get { return version; } set { version = value; } }  // The "vendor" static property public static string Vendor { get { return vendor; } set { vendor = value; } }  // ... More (non)static code here ...} |

В този пример сме избрали да пазим стойността на статичните свойства в статични променливи (което е логично, тъй като те са обвързани само с класа). Свойствата, които разглеждаме са съответно версия (**Version**) и произво­ди­тел (**Vendor**). За всяко едно от тях сме създали статични методи за четене и модификация. Така всички обекти от този клас, ще могат да извлекат текущата версия и производителя на системата, която описва класа. Съответно, ако някой ден бъде направено обновление на версията на системата например стойността стане **0.2**, всеки от обектите, ще получи като резултат новата версия, чрез достъпване на свойството на класа.

**Статичните свойства и ключовата дума this**

Подобно на статичните методи, в статичните свойства не може да се използва ключовата дума **this**, тъй като статичното свойство е асоциира­но единствено с класа, и не "разпознава" обектите от даден клас.

|  |  |
| --- | --- |
| clip_image007[18] | **В статичните свойства не може да се използва ключовата дума this.** |

**Достъп до статични свойства**

По подобие на статичните полета и методи, статичните свойства могат да бъдат достъпвани чрез точкова нотация приложена единствено към името на класа, в който са декларирани.

За да се уверим, нека се опитаме да достъпим свойството **Version** през променлива от класа **SystemInfo**:

|  |
| --- |
| public static void Main(){ SystemInfo sysInfoInstance = new SystemInfo(); Console.WriteLine("System version: " +  sysInfoInstance.Version);} |

При опит за компилация на горния код, получаваме следното съобщение за грешка:

|  |
| --- |
| Member 'SystemInfo.Version.get' cannot be accessed with an instance reference; qualify it with a type name instead |

Съответно, ако се опитаме да достъпим статичните свойства чрез името на класа, кодът се компилира и работи правилно:

|  |
| --- |
| public static void Main(){ // Invocation of static property setter SystemInfo.Vendor = "Microsoft Corporation";  // Invocation of static property getters Console.WriteLine("System version: " + SystemInfo.Version); Console.WriteLine("System vendor: " + SystemInfo.Vendor);} |

Кодът се компилира и резултатът от изпълнението му е:

|  |
| --- |
| System version: 0.1System vendor: Microsoft Corporation |

Преди да преминем към следващата секция, нека обърнем внимание на отпечатаната стойност на свойството **Vendor**. Тя е "**Microsoft Corpora­tion**", въпреки че в класа **SystemInfo** сме я инициализирали със стой­ността "**Microsoft**". Това е така, тъй като променихме стойността на свойството **Vendor** на първия ред от метода **Main()**, чрез извикване на метода му за модификация.

|  |  |
| --- | --- |
| clip_image007[19] | **Статичните свойства могат да бъдат достъпвани единстве­но чрез точкова нотация, приложена към името на класа, в който са декларирани.** |

**Статични класове**

За пълнота трябва да обясним, че можем да декларираме класовете като статични. Подобно на статичните членове, един клас е статичен, когато при декларацията му е използвана ключовата дума **static**:

|  |
| --- |
| **[<modifiers>] static class <class\_name>****{** // ... Class body goes here**}** |

Когато един клас е деклариран като статичен, това е индика­ция, че този клас съдържа само статични членове (т.е. статични полета, методи, свойства) и не може да се инстанцира.

Употребата на статични класове е рядка и най-често е свързана с употребата на статични методи, които не принадлежат на нито един конкретен обект. По тази причина, подробностите за статичните класове излизат извън обсега на тази книга. Любознателният читател може да намери повече информация на сайта на Microsoft (MSDN).

**Статични конструктори**

За да приключим със секцията за статичните членове на класа, трябва да споменем и класовете могат да имат и **статичен конструктор** (т.е. конструктор, които има ключовата дума **static** в декларацията си):

|  |
| --- |
| **[<modifiers>] static <class\_name>([<parameters\_list>])****{****}** |

Статични конструктори могат да бъдат декларирани, както в статични, така и в нестатични класове. Те се изпълняват само веднъж, когато първото от следните две събития се случи за първи път:

1.  Създава се обект от класа.

2.  Достъпен е статичен елемент от класа (поле, метод, свойство).

Най-често статичните конструктори се използват за инициализацията на статични полета.

**Статичен конструктор – пример**

Да разгледаме един пример за използването на статичен конструктор. Искаме да направим клас, който изчислява бързо корен квадратен от цяло число и връща цялата част на резултата – също цяло число. Тъй като изчисля­ването на корен квадратен е времеотнемаща математическа операция, включваща пресмятания с реални числа и изчисляване на сходящи редове, е добра идея тези изчисления да се изпълнят еднократно при стартиране на програмата, а след това да се използват вече изчислени стойности. Разбира се, за да се направи такова предварително изчисление (precomputation) на всички квадратни корени в даден диапазон, трябва първо да се дефинира този диапазон и той не трябва да е прекалено широк (например от 1 до 1000). След това е необходимо при първо поиск­ване на корен квадратен на дадено число да се преизчислят всички квадратни корени в дадения диапазон, а след това да се върне вече готовата изчислена стойност. При следващо поискване на корен квадра­тен, всички стойности в дадения диапазон са вече изчислени и се връщат директно. Ако пък никога в програмата не се изчислява корен квадратен, предварителните изчисления трябва изобщо да не се изпълнят.

Чрез описания подход първоначално се инвестира някакво процесорно време за предварителни изчисления, но след това извличането на корен квадратен се извършва изключително бързо. Ако изчисляването на корен квадратен се извършва многократно, преизчислението ще увеличи значи­телно производителността.

Всичко това може да се имплементира в един статичен клас със статичен конструктор, в който да се преизчисляват квадратните корени. Вече изчислените резултати могат да се съхраняват в статичен масив. За извличане на вече преизчислена стойност може да се използва статичен метод. Тъй като предварителните изчисления се извършват в статичния конструктор, ако класът за преизчислен корен квадратен не се използва, те няма да се извършат и ще се спести процесорно време и памет. Ето как би могла да изглежда имплементацията:

|  |
| --- |
| static class SqrtPrecalculated{ public const int MaxValue = 1000;  // Static field  private static int[] sqrtValues;   // Static constructor  static SqrtPrecalculated() { sqrtValues = new int[MaxValue + 1]; for (int i = 0; i < sqrtValues.Length; i++) { sqrtValues[i] = (int)Math.Sqrt(i); } }  // Static method  public static int GetSqrt(int value) { if ((value < 0) || (value > MaxValue)) { throw new ArgumentOutOfRangeException(String.Format( "The argument should be in range [0..{0}].", MaxValue)); } return sqrtValues[value]; }} class SqrtTest{ static void Main() { Console.WriteLine(SqrtPrecalculated.GetSqrt(254)); // Result: 15 }} |

**Изброени типове (enumerations)**

По-рано в тази глава ние разгледахме какво представляват [константите](http://www.introprogramming.info/intro-csharp-book/read-online/glava14-definirane-na-klasove/#_%D0%9A%D0%BE%D0%BD%D1%81%D1%82%D0%B0%D0%BD%D1%82%D0%B8_%28constants%29), как се декларират и как се използват. В тази връзка, сега ще разгледаме една конструкция от езика С#, при която можем множество от константи, които са свързани логически, да ги свържем и чрез средствата на езика. Това средство на езика са така наречените изброени типове.

**Декларация на изброените типове**

**Изброен тип (enumeration)** наричаме конструкция, която наподобява клас, но с тази разлика, че в тялото на класа можем да декларираме само константи. Изброените типове могат да приемат стойности само измежду изброените в типа константи. Променлива от изброен тип може да има за стойност някоя измежду изброените в типа стойности (константи), но не може да има стойност **null**.

Формално казано, изброените типове се декларират с помощта на запазената дума **enum** вместо **class**:

|  |
| --- |
| **[<modifiers>] enum <enum\_name>****{** **constant1 [, constant2 [, [, ... [, constantN]]****}** |

Под **<modifiers>** разбираме модификаторите за достъп **public**, **internal** и **private**. Идентификаторът **<enum\_name>** следва правилата за имена на класове в С#. В блока на изброения тип се декларират константите, разделени със запетайки.

Нека разгледаме един пример. Да дефинираме изброен тип за дните от седмицата (ще го наречем **Days**). Както се досещаме, константите, които ще се съдържат в този изброен тип са имената на дните от седмицата:

|  |
| --- |
| **Days.cs** |
| enum Days{ Mon, Tue, Wed, Thu, Fri, Sat, Sun} |

Именуването на константите в един изброен тип следва правилото за именуване на константи, което обяснихме в секцията "[Именуване на константите](http://www.introprogramming.info/intro-csharp-book/read-online/glava14-definirane-na-klasove/#_%D0%98%D0%BC%D0%B5%D0%BD%D1%83%D0%B2%D0%B0%D0%BD%D0%B5_%D0%BD%D0%B0_%D0%BA%D0%BE%D0%BD%D1%81%D1%82%D0%B0%D0%BD%D1%82%D0%B8%D1%82%D0%B5)".

Трябва да отбележим, че всяка една от константите в изброения тип е от тип този изброен тип, т.е. в нашия пример **Mon** e от тип **Days**, както и всяка една от останалите константи.

С други думи, ако изпълним следния ред:

|  |
| --- |
| Console.WriteLine(Days.Mon is Days); |

ще бъде отпечатан резултат:

|  |
| --- |
| True |

Нека повторим:

|  |  |
| --- | --- |
| clip_image007[20] | **Изброените типове са множество от константи от тип – този изброен тип.** |

**Същност на изброените типове**

Всяка една константа, която е декларирана в един изброен тип, е асоциирана с някакво цяло число. По подразбиране, за това целочислено скрито представяне на константите в един изброен тип се използва **int**.

За да покажем "целочислената природа" на константите в изброените типове, нека се опитаме да разберем какво е численото представяне на константата отговаряща на "понеделник" от примера от предходната подсекция:

|  |
| --- |
| int mondayValue = (int)Days.Mon;Console.WriteLine(mondayValue); |

След като го изпълним, резултатът ще бъде:

|  |
| --- |
| 0 |

Стойностите, асоциирани с константите в един изброен тип по подразби­ране са индексите в списъка с константи на този тип, т.е. числата от 0 до броя константи в типа минус единица. Така, ако разгледаме примера с изброения тип за дните в седмицата, използван в предходната подсекция, константата **Mon** е асоциирана с числената стойност 0, константата **Tue** с целочислената стойност 1, **Wed** – с 2, и т.н.

|  |  |
| --- | --- |
| clip_image007[21] | **Всяка константа в един изброен тип реално е текстово представяне на някакво цяло число. По подразбиране, това число е индексът на константата в списъка от кон­стан­ти на изброения тип.** |

Въпреки целочислената природа на константите в един изброен тип, когато се опитаме да отпечатаме дадена константа, ще бъде отпечатано текстовото й представяне зададено при декларацията й:

|  |
| --- |
| Console.WriteLine(Days.Mon); |

След като изпълним горния код, резултатът ще бъде следният:

|  |
| --- |
| Mon |

**Скрита числена стойност на константите в изброени типове**

Както вече се досещаме, можем да променим **числената стойност на константите в един изброен тип**. Това става като по време на декла­рацията присвоим стойността, която предпочитаме, на всяка една от константите.

|  |
| --- |
| **[<modifiers>] enum <enum\_name>****{** **constant1[=value1] [, constant2[=value2] [, ... ]]****}** |

Съответно **value1**, **value2**, и т.н. трябва да са цели числа.

За да добием по-ясна представа за току-що дадената дефиниция, нека разгледаме следния пример: нека имаме клас **Coffee**, който представя чаша кафе, която клиентите поръчват в някакво заведение:

|  |
| --- |
| **Coffee.cs** |
| public class Coffee{ public Coffee() { }} |

В това заведение, клиентът може да поръча различно количество кафе, като кафе-машината има предефинирани стойности – "малко" – 100 ml, "нормално" – 150 ml и "двойно" – 300 ml. Следователно, можем да си декларираме един изброен тип **CoffeSize**, който има съответно три константи – **Small**, **Normal** и **Double**, на които ще присвоим съответства­щите им количества:

|  |
| --- |
| **CoffeeSize.cs** |
| public enum CoffeeSize { Small=100, Normal=150, Double=300} |

Сега можем да добавим поле и свойство към класа **Coffee**, които отра­зяват какъв тип кафе си е поръчал даден клиент:

|  |
| --- |
| **Coffee.cs** |
| public class Coffee{ public CoffeeSize size;  public Coffee(CoffeeSize size) { this.size = size; }  public CoffeeSize Size { get { return size; } }} |

Нека се опитаме да отпечатаме стойностите на количеството кафе за едно нормално кафе и за едно двойно:

|  |
| --- |
| static void Main(){ Coffee normalCoffee = new Coffee(CoffeeSize.Normal); Coffee doubleCoffee = new Coffee(CoffeeSize.Double);  Console.WriteLine("The {0} coffee is {1} ml.", normalCoffee.Size, (int)normalCoffee.Size); Console.WriteLine("The {0} coffee is {1} ml.", doubleCoffee.Size, (int)doubleCoffee.Size);} |

Како компилираме и изпълним този метод, ще бъде отпечатано следното:

|  |
| --- |
| The Normal coffee is 150 ml.The Double coffee is 300 ml. |

**Употреба на изброените типове**

Основната цел на изброените типове е да заменят числените стойности, които бихме използвали, ако не съществуваха изброените типове. По този начин, кодът става по-изчистен и по-лесен за четене.

Друго много важно приложение на изброените типове е принудата от страна на компилатора да бъдат използвани константите от изброения тип, а не просто числа. По този начин ограничаваме максимално бъдещи грешки в кода. Например, ако изпол­зваме променлива от тип **int** вместо от изброен тип и набор константи за валидните стойности, нищо не пречи да присвоим на променливата примерно -6723.

За да стане по-ясно, нека разгледаме следния пример: да създадем клас, който представлява калкулатор за пресмятане на цената на всеки от видовете кафе, които се предлагат в заведението:

|  |
| --- |
| **PriceCalculator.cs** |
| public class PriceCalculator{ public const int SmallCoffeeQuantity = 100; public const int NormalCoffeeQuantity = 150; public const int DoubleCoffeeQuantity = 300;  public CashMachine() { }  public double CalcPrice(int quantity) { switch (quantity) { case SmallCoffeeQuantity: return 0.20; case NormalCoffeeQuantity: return 0.30; case DoubleCoffeeQuantity: return 0.60; default: throw new InvalidOperationException( "Unsupported coffee quantity: " + quantity); } }} |

Създали сме три константи, отразяващи вместимостта на чашките за кафе, които имаме в заведението, съответно 100, 150 и 300 ml. Освен това **очакваме**, че потребителите на нашия клас ще използват прилежно дефинираните от нас константи, вместо числа – **SmallCoffeeQuantity**, **NormalCoffeeQuantity** и **DoubleCoffeeQuantity**. Методът **CalcPrice(int)** връща съответната цена, като я изчислява според подаденото количество.

Проблемът, се състои в това, че някой може да реши да не използва дефинираните от нас константи и може да подаде като параметър на нашия метод невалидно число, например -1 или 101. В този случай, ако методът не прави проверка за невалидно количество, най-вероятно ще върне грешна цена, което е некоректно поведение.

За да избегнем този проблем, ще използваме една особеност на изброени­те типове, а именно, че константите в изброените типове могат да се използват в конструкции **switch-case**. Те могат да бъдат подавани като стойност на оператора **switch** и съответно – като операнди на оператора **case**.

|  |  |
| --- | --- |
| clip_image007[22] | **Константите на един изброен тип могат да бъдат изпол­звани в конструкции switch-case.** |

Нека преработим метода за получаване на цената за чашка кафе в зависимост от вместимостта на чашката, като този път използваме изброения тип **CoffeeSize**, който декларирахме в предходните примери:

|  |
| --- |
| public double getPrice(CoffeeSize coffeeSize){ switch (coffeeSize) { case CoffeeSize.Small:  return 0.20; case CoffeeSize.Normal: return 0.40; case CoffeeSize.Double: return 0.60; default:  throw new InvalidOperationException( "Unsupported coffee quantity: " +((int)coffeeSize)); }} |

Както виждаме, в този пример възможността потребителите на нашия метод да провокират непредвидено поведение на метода е нищожна, тъй като ги принуждаваме да използват точно определени стойности, които да подадат като аргументи, а именно константите на изброения тип **CoffeeSize**. Това е едно от предимствата на константите, декларирани в изброени типове пред константите декларирани в произволен клас.

|  |  |
| --- | --- |
| clip_image007[23] | **Винаги, когато съществува възможност, използвайте из­броен тип вместо множество константи де­кларирани в някакъв клас.** |

Преди да приключим секцията за изброените типове, трябва да споменем, че изброените типове трябва да се използват много внимателно при работа с конструкцията **switch-case**. Например, ако някой ден, собстве­никът на заведението купи много големи чаши за кафе, ще трябва да добавим нова константа в списъка с константи на изброения тип **CoffeeSize**, нека я наречем **Overwhelming**:

|  |
| --- |
| **CoffeeSize.cs** |
| public enum CoffeeSize { Small=100, Normal=150, Double=300, Overwhelming=600} |

Когато се опитаме да пресметнем цената на кафе с новото количество, методът, който пресмята цената, ще хвърли изключение, което съобщава на потребителя, че такова количество кафе не се предлага в заведението.

Това, което трябва да направим, за да решим този проблем е да добавим ново **case**-условие, което да отразява новата константа в изброения тип **CoffeeSize**.

|  |  |
| --- | --- |
| clip_image007[24] | **Когато модифицираме списъка с константите на вече съществуващ изброен тип, трябва да внимаваме, да не нарушим логиката на кода, който вече съществува и използва декларираните до момента константи.** |

**Вътрешни класове (nested classes)**

В C# **вътрешен (nested)** се нарича клас, който е деклариран вътре в тялото на друг клас. Съответно, клас, който обвива вътрешен клас се нарича **външен клас (outer class)**.

Основните причини да се декларира един клас в друг са следните:

1.  За по-добра организация на кода, когато работим с обекти от реалния свят, между които има специална връзка и единият не може да съществува без другия.

2.  Скриване на даден клас в друг клас, така че вътрешният клас да не бъде използван извън обвиващия го клас.

По принцип вътрешни класове се ползват рядко, тъй като те усложняват структурата на кода и увеличават нивата на влагане.

**Декларация на вътрешни класове**

Вътрешните класове се декларират по същия начин, както нормалните класове, но се разполагат вътре в друг клас. Позволените модификатори в декларацията на класа са следните:

1.  **public** – вътрешният клас е достъпен от кое да е асембли.

2.  **internal** – вътрешният клас е достъпен в текущото асембли, в което се намира външния клас.

3.  **private** – достъпът е ограничен само до класа, който съдържа вътрешния клас.

4.  **static** – вътрешният клас съдържа само статични членове.

Има още четири позволени модификатора – **abstract**, **protected**, **protected internal**, **sealed** и **unsafe**, които са извън обхвата и темати­ката на тази глава и няма да бъдат разглеждани тук.

Ключовата дума **this** за един вътрешен клас, притежава връзка единствено към вътрешния клас, но не и към външния. Полетата на външния клас **не могат** да бъдат достъпвани използвайки референцията **this**. Ако е необходимо полетата на външния клас да бъдат достъпвани от вътрешния, трябва при създаването на вътрешния клас да се подаде референция към външния клас.

Статичните членове (полета, методи, свойства) на външния клас са достъпни от вътрешния независимо от нивото си на достъп.

**Вътрешни класове – пример**

Нека разгледаме следния пример:

|  |
| --- |
| **OuterClass.cs** |
| public class OuterClass{ private string name;  private OuterClass(string name) { this.name = name; }  private class NestedClass { private string name; private OuterClass parent;  public InnerClass(OuterClass parent, string name) { this.parent = parent; this.name = name; }  public void PrintNames() { Console.WriteLine("Nested name: " + this.name); Console.WriteLine("Outer name: " + this.parent.name); } }  public static void Main() { OuterClass outerClass = new OuterClass("outer"); NestedClass nestedClass = new OuterClass.InnerClass(outerClass, "nested"); nestedClass.PrintNames(); }} |

В примера външният клас **OuterClass** дефинира в себе си като член класа **InnerClass**. Нестатичните методи на вътрешния клас имат достъп както до собствената си инстанция **this**, така и до инстанцията на външния клас **parent** (чрез синтаксиса **this.parent**, ако референцията **parent** е доба­вена от програмиста). В примера при създаването на вътрешния клас на конструктора му се подава **parent** референцията на външния клас.

Ако изпълним горния пример, ще получим следния резултат:

|  |
| --- |
| Inner name: innerOuter name: outer |

**Употреба на вътрешни класове**

Нека разгледаме един пример. Нека имаме клас за кола – **Car**. Всяка кола има двигател, както и врати. За разлика от вратите на колата обаче, двигателят няма смисъл разглеждан като елемент извън колата, тъй като без него, колата не може да работи, т.е. имаме композиция (вж. секция "Композиция" в глава "[Принципи на обектно-ориентираното програмиране](http://www.introprogramming.info/intro-csharp-book/read-online/glava20-principi-na-obektno-orientiranoto-programirane)").

|  |  |
| --- | --- |
| clip_image007[25] | **Когато връзката между два класа е композиция, класът, който логически е част от друг клас, е удобно да бъде деклариран като вътрешен клас.** |

Следователно, ако декларираме класа за кола **Car**, ще е подходящо да си създадем като вътрешен клас **Engine**, който ще отразява съответно концепцията за двигател на колата:

|  |
| --- |
| **Car.cs** |
| class Car{ Door FrontRightDoor; Door FrontLeftDoor; Door RearRightDoor; Door RearLeftDoor; Engine engine;  public Car() { engine = new Engine(); engine.horsePower = 2000; }  public class Engine { public int horsePower; }} |

**Декларация на изброен тип в клас**

Преди да преминем към следващата тема за шаблонните типове (generics), трябва да отбележим, че понякога изброените типове се налага и **могат** да бъдат декларирани в рамките на даден клас с оглед на по-добрата капсулация на класа.

Например, изброеният тип **CoffeeSize**, който създадохме в предходната секция може да бъде деклариран вътре в тялото на класа **Coffee**, като по този начин се подобрява капсулацията:

|  |
| --- |
| **Coffee.cs** |
| class Coffee{ // Enumeration public static enum CoffeeSize  {  Small = 100, Normal = 150, Double = 300  }  // Instance variable private CoffeeSize size;  public Coffee(CoffeeSize size) { this.size = size; }  public CoffeeSize Size { get { return size; } }} |

Съответно, методът за изчисляване на цената на едно кафе ще претърпи лека модификация:

|  |
| --- |
| public double CalcPrice(Coffee.CoffeeSize coffeeSize){ switch (coffeeSize) { case Coffee.CoffeeSize.Small: return 0.20;  case Coffee.CoffeeSize.Normal: return 0.40;  case Coffee.CoffeeSize.Double: return 0.60;  default: throw new InvalidOperationException( "Unsupported coffee quantity: " + ((int)coffeeSize)); }} |

**Шаблонни типове и типизиране (generics)**

В тази секция ще обясним концепцията за типизиране на класове. Преди да започнем, обаче, нека разгледаме един пример, който ще ни помогне за разберем по-лесно идеята.

**Приют за бездомни животни – пример**

Нека имаме два класа. Нека класът **Dog** описва куче:

|  |
| --- |
| **Dog.cs** |
| public class Dog{} |

И нека класът **Cat** описва котка:

|  |
| --- |
| **Cat.cs** |
| public class Cat{} |

След това искаме да си създадем клас, който описва приют за бездомни животни – **AnimalShelter**. Този клас има определен брой свободни клетки, който определя броя на животни, които могат да намерят подслон в приюта. Особеното на класа, който искаме да създадем е, че той трябва да подслонява само животни от един и същ вид, в нашия случай или само кучета, или само котки, защото съвместното съжителство на различни видове животни не винаги е добра идея.

Ако се замислим как ще решим задачата със знанията, които имаме до момента, стигаме до извода, че за да гарантираме, че нашият клас ще съдържа елементи само от един тип, трябва да използваме масив от еднакви обекти. Тези обекти може да са кучета, котки или просто инстан­ции на универсалния тип **object**.

Например, ако искаме да направим приют за кучета, ето как би изглеждал нашият клас:

|  |
| --- |
| **AnimalsShelter.cs** |
| public class AnimalShelter{ private const int DefaultPlacesCount = 20;  private Dog[] animalList; private int usedPlaces;  public AnimalShelter() : this(DefaultPlacesCount) { }  public AnimalShelter(int placesCount) { this.animalList = new Dog[placesCount]; this.usedPlaces = 0; }  public void Shelter(Dog newAnimal) { if (this.usedPlaces >= this.animalList.Length) { throw new InvalidOperationException("Shelter is full."); } this.animalList[this.usedPlaces] = newAnimal; this.usedPlaces++; }  public Dog Release(int index) { if (index < 0 || index >= this.usedPlaces) { throw new ArgumentOutOfRangeException( "Invalid cell index: " + index); } Dog releasedAnimal = this.animalList[index]; for (int i = index; i < this.usedPlaces - 1; i++) { this.animalList[i] = this.animalList[i + 1]; } this.animalList[this.usedPlaces - 1] = null; this.usedPlaces--;  return releasedAnimal; }} |

Капацитетът на приюта (броят животни, които могат да се приютят в него) се задава при създаване на обекта. По подразбиране е стойността на константата **DefaultPlacesCount**. Полето **usedPlaces** използва­ме за следене на заетите до момента клетки (едновременно с това го из­ползваме за индекс в масива, да "сочим" към първото свободно място отляво на дясно в масива).



Създали сме метод за добавяне на ново куче в приюта – **Shelter(…)** и съот­вет­но за освобождаване от приюта – **Release(int)**.

Методът **Shelter()** добавя всяко ново животно в първата свободна клетка в дясната част на масива (ако има такава).

Методът **Release(int)** приема номера на клетката, от която ще бъде освободено куче (т.е. номера на индекса в масива, където е съхранена връзка към обекта от тип **Dog**).



След това премества животните, които се намират в клетки с по-голям номер от номера на клетката, от която ще извадим едно куче, с една позиция на наляво (стъпки 2 и 3 на схемата по-долу).



Освободената клетка на позиция **usedPlaces-1** се маркира като свободна, като й се присвоява стойност **null**. Това осигурява освобождаването на референцията към нея и съответно позволява на системата за почистване на паметта (garbage collector) да освободи обекта, ако той не се ползва никъде другаде в програмата в същия момент. Това предпазва недиректна загуба на памет (memory leak).

Накрая присвоява на полето **usedPlaces** номера на последната свобод­на клетка (стъпки 4 и 5 от схемата отгоре).



Забелязва се, че "изваждането" на животно от дадена клетка би могло да е бавна операция, тъй като изисква прехвърляне на животните от следващите клетки с една позиция наляво. В главата "[Линейни структури от данни](http://www.introprogramming.info/intro-csharp-book/read-online/glava16-lineini-strukturi-ot-danni)" ще разгледаме и по-ефективни начини за предста­вяне на приюта за животни, но за момента нека се фокусираме върху темата за шаблонните типове.

До този момент успяхме да имплементираме функционалността на приюта - класът **AnimalShelter**. Когато работим с обекти от тип **Dog**, всичко се компилира и изпълнява безпроблемно:

|  |
| --- |
| public static void Main(){ AnimalShelter dogsShelter = new AnimalShelter(10); Dog dog1 = new Dog(); Dog dog2 = new Dog(); Dog dog3 = new Dog();  dogsShelter.Shelter(dog1); dogsShelter.Shelter(dog2); dogsShelter.Shelter(dog3);  dogsShelter.Release(1); // Releasing dog2} |

Какво ще стане, обаче, ако се опитаме да използваме класа **AnimalShelter** за обекти от тип **Cat**:

|  |
| --- |
| public static void Main(){ AnimalShelter dogsShelter = new AnimalShelter(10);  Cat cat1 = new Cat();  dogsShelter.Shelter(cat1); } |

Както се очаква, компилаторът извежда грешка:

|  |
| --- |
| The best overloaded method match for 'AnimalShelter.Shelter( Dog)' has some invalid arguments. Argument 1: cannot convert from 'Cat' to 'Dog' |

Следователно, ако искаме да направим приют за котки, няма да успеем да преизползваме вече създадения от нас клас, въпреки, че операциите по добавяне и изваждане на животни от приюта ще бъдат идентични. Следователно, буквално ще трябва да копираме класа **AnimalShelter** и да променим само типа на обектите, с които се работи – **Cat**.

Да, но ако решим да правим приют и за други видове животни? Колко класа за приюти за конкретния тип животни ще трябва да създадем?

Виждаме, че това решение на задачата не е достатъчно изчерпателно и не изпълнява изцяло условията, които си бяхме поставили, а именно – да съществува **един единствен клас**, който описва нашия приют за каквито и да е животни (т.е. за всякакви обекти) и при работа с него той да **съдържа само един вид животни** (т.е. единствено обекти от един и същ тип).

Бихме могли да използваме вместо типа **Dog** универсалния тип **object**, който може да приема като стойности **Dog**, **Cat** и всякакви други типове данни, но това ще създаде някои неудобства, свързани с нуждата от обратно преобразуване от **object** към **Dog**, когато се прави приют за кучета, а той съдържа клетки от тип object вместо от тип **Dog**.

За да решим задачата ефективно се налага да използваме една функцио­налност на езика С#, която ни позволява да удовлетворим всички условия едновременно. Тя се нарича **шаблонни класове (generics)**.

**Какво представляват шаблонните класове?**

Както знаем, когато за работата на един метод е нужна допълнителна информация, тази информация се подава на метода чрез параметри. По време на изпълнение на програмата, при извикване на метода, подаваме аргументи на метода, те се присвояват на параметри­те му и след това се използват в тялото на метода.

По подобие на методите, когато знаем, че функционалността (действията) капсулирана в един клас, може да бъде приложена не само към обекти от един, а от много (разнородни) типове, и тези типове не са известни по време на деклариране на класа, можем да използваме една функционал­ност на езика С# наречена **шаблонни типове (generics)**. Тя ни позволява да декларираме параметри на самия клас, чрез които обознача­ваме неиз­вестния тип, с който класът ще работи в последствие. След това, когато инстанцираме нашия типизиран клас, ние заместваме неизвестния тип с конкретен. Съответно новосъздаденият обект ще работи само с обекти от конкретния тип, който сме задали при инициализацията му. Конкретният тип може да бъде всеки един клас, който компилаторът разпознава, включително структура, изброен тип или друг шаблонен клас.

За да добием по-ясна представа за същността на шаблонните типове, нека се върнем към нашата задача от предходната секция. Както се досещаме, кла­сът, който описва приют на животни (**AnimalShelter**), може да опе­рира с различни типове животни. Следователно, ако искаме да създа­дем генерално решение на задачата, по време на декларация на класа **AnimalShelter**, ние не можем да знаем какъв тип животни ще бъдат приютявани в приюта. Това е достатъчна индикация, че можем да типизираме нашия клас, добавяйки към декларацията на класа, като параметър, неизвестния ни тип на животни.

В последствие, когато искаме да създадем приют за кучета например, на този параметър на класа ще подадем името на нашия тип – класа **Dog**. Съответно, ако създаваме приют за котки, ще подадем типа **Cat** и т.н.

|  |  |
| --- | --- |
| clip_image007[26] | **Типизирането на клас (създаването на шаблонен клас) представлява добавяне, към декларацията на един клас, на параме­тър (за­ме­стител) на неизвестен тип, с който класът ще работи по време на изпълне­ние на програмата. В послед­ствие, когато класът бива инстанциран, този параметър се замества с името на някой кон­кретен тип.** |

В следващите секции ще се запознаем със синтаксиса на типизирането на класове и ще представим нашия пример преработен, така че да използва типизиране.

**Декларация на типизиран (шаблонен) клас**

Формално, типизирането на класове се прави, като към декларацията на класа, след самото име на класа се добави **<T>**, където **T** е заместителят (параметърът) на типа, който ще се използва в последствие:

|  |
| --- |
| **[<modifiers>] class <class\_name><T>****{****}** |

Трябва да отбележим, че знаците '**<**' и '**>**', които ограждат заместителя **T** са задължителна част от синтаксиса на езика С# и трябва да участват в декларацията на типизирането на даден клас.

Декларацията на типизирания клас, описващ приюта за бездомни живот­ни, би изглеждала по следния начин:

|  |
| --- |
| class AnimalShelter<T>{ // Class body here ...} |

По този начин, можем да си представим, че правим шаблон на нашия клас **AnimalShelter**, който в последствие ще конкретизираме, заменяйки **T** с конкретен тип, например **Dog**.

Eдин клас може да има и повече от един замести­тел (да е параметризиран по повече от един тип), в зависимост от нуждите му:

|  |
| --- |
| **[<modifiers>] class <class\_name><T1 [, T2, [... [, Tn]>****{****}** |

Ако класът се нуждае от няколко различни неизвестни типа, тези типове трябва да се изброят, чрез запетайка между знаците '**<**' и '**>**' в деклараци­ята на класа, като всеки един от използваните заместители трябва да е различен идентификатор (например различна буква) – в дефиницията са указани като **T1**, **T2**, ..., **Тn**.

В случай, че искахме да създадем приют за животни от смесен тип, такъв че да приютява кучета и котки едновременно, можехме да декларираме нашия клас по следния начин:

|  |
| --- |
| class AnimalShelter<T, U>{ // Class body here ...} |

Ако това беше нашия случай, щяхме да използваме първия параметър **T**, за означаване на обектите от тип **Dog**, с които нашия клас щеше да оперира и **U** – за означаване на обектите от тип **Cat**.

**Конкретизиране на типизирани класове**

Преди да представим повече подробности за типизацията, нека поглед­нем как се използват типизираните класове. Използването на типизирани класове става по следния начин:

|  |
| --- |
| **<class\_name><concrete\_type> <variable\_name> =****new <class\_name><concrete\_type>();** |

Отново, подобно на заместителя **T** в декларацията на нашия клас, знаците '**<**' и '**>**', които ограждат кон­кретния клас **concrete\_type**, са задължителни.

Ако искаме да създадем два приюта, един за кучета и един за котки, ще трябва да използваме следния код:

|  |
| --- |
| AnimalShelter<Dog> dogsShelter = new AnimalShelter<Dog>();AnimalShelter<Cat> catsShelter = new AnimalShelter<Cat>(); |

По този начин сме сигурни, че приютът **dogsShelter** винаги ще съдържа обекти от тип **Dog**, а променливата **catsShelter** ще оперира винаги с обекти от тип **Cat**.

**Използване на неизвестните типове в декларация на полета**

Веднъж използвани по време на декларацията на класа, параметрите, които са използвани за указване на неизвестните типове са видими в цялото тяло на класа, следователно могат да се използват за деклариране на полета както всеки друг тип:

|  |
| --- |
| **[<modifiers>] T <field\_name>;** |

Както можем да се досетим, в нашия пример с приюта за бездомни животни, можем да използваме тази възможност на езика С#, за да декларираме типа на полето **animalsList**, в което съхраняваме референ­ции към обектите на приютените животни, вместо с конкретния тип **Dog**, с параметъра **Т**:

|  |
| --- |
| private T[] animalList; |

За сега нека приемем, че когато създаваме обект от нашия клас, подавайки конкретен тип (например **Dog**), по време на изпълнение на програмата неизвестният тип **Т** ще бъде заменен с въпросния тип. Ако сме избрали да създадем приют за кучета, можем да смятаме, че нашето поле е декла­рирано по следния начин:

|  |
| --- |
| private Dog[] animalList; |

Съответно, когато искаме да инициализираме въпросното поле в кон­структора на нашия клас, ще трябва да го направим по същия начин, както обикновено – създаваме масив, само че използвайки заместителя на неизвестния тип – **Т**:

|  |
| --- |
| public AnimalShelter(int placesNumber){ animalList = new T[placesNumber]; // Initialization usedPlaces = 0;} |

**Използване на неизвестните типове в декларация на методи**

Тъй като един неизвестен тип, използван в декларацията на един типизи­ран клас е видим от отварящата до затварящата скоба на тялото на класа, освен за декларация на полета, той може да бъде използван и в деклара­цията на методи, а именно:

-     Като параметър в списъка от параметри на метода:

|  |
| --- |
| **<return\_type> MethodWithParamsOfT(T param)** |

-     Като резултат от изпълнението на метода:

|  |
| --- |
| **Т MethodWithReturnTypeOfT(<params>)** |

Както вече се досещаме, използвайки нашия пример, можем да адаптира­ме методите **Shelter** и **Release**, съответно:

-     Като метод с параметър от неизвестен тип **Т**:

|  |
| --- |
| public void Shelter(T newAnimal){ // Method's body goes here ...} |

-     И метод, който връща резултат от неизвестен тип **Т**:

|  |
| --- |
| public T Release(int i){ // Method's body goes here ...} |

Както вече знаем, когато създадем обект от нашия клас приют и неиз­вестния тип го заменим с някой конкретен тип (например **Cat**), по време на изпълнение на програмата горните методи ще имат следния вид:

-     Параметърът на метода **Shelter** ще бъде от тип **Cat**:

|  |
| --- |
| public void Shelter(Cat newAnimal){ // Method's body goes here ...} |

-     Методът **Release** ще връща резултат от тип **Cat**:

|  |
| --- |
| public Cat Release(int i){ // Method's body goes here ...} |

**Типизирането (generics) зад кулисите**

Преди да продължим, нека обясним какво става в паметта на компютъра, когато работим с типизирани класове.



Първо декларираме нашия типизиран клас **MyClass<T>** (generic class description в горната схема). След това компилаторът транслира нашия код на междинен език (MSIL), като транслираният код, съдържа инфор­мация, че класът е типизиран, т.е. работи с неопределени до момента типове. По време на изпълнение, когато някой се опитва да работи с нашия типизиран клас и да го използва с конкретен тип, се създава ново **описание на клас** (concrete type class description в схемата по-горе), което е идентично с това на типизирания клас, с тази разлика, че навсякъде където е използвано **T**, сега се заменя с конкретния тип. Например ако се опитаме да използваме **MyClass<int>**, навсякъде, където в нашия код e използван неизвестния параметър **T**, ще бъде заме­нен с **int**. Едва след това, можем да създадем обект от типизирания клас с конкретен тип **int**. Особеното тук е, че за да се създаде този обект, ще се използва описанието на класа, което бе създадено междувременно (concrete type class description). Инстанцирането на шаблонен клас по дадени конкретни типове на неговите параметри се нарича "**специали­зация на тип**" или "**разгъване на шаблонен клас**".

Използвайки нашия пример, ако създадем обект от тип **AnimalShelter<T>**, който работи само с обекти от тип **Dog**, ако се опитаме да добавим обект от тип **Cat**, това ще доведе до грешка при компилация почти идентична с грешките, които бяха изведени при опит за добавяне на обект от тип **Cat**, към обект от тип **AnimalShelter**, който създадохме в първата подсекция "[Приют за бездомни животни – пример](http://www.introprogramming.info/intro-csharp-book/read-online/glava14-definirane-na-klasove/#_%D0%9F%D1%80%D0%B8%D1%8E%D1%82_%D0%B7%D0%B0_%D0%B1%D0%B5%D0%B7%D0%B4%D0%BE%D0%BC%D0%BD%D0%B8)":

|  |
| --- |
| public static void Main(){ AnimalShelter<Dog> dogsShelter = new AnimalShelter<Dog>(10);  Cat cat1 = new Cat();  dogsShelter.Shelter(cat1);} |

Както се очакваше, получаваме следните съобщения:

|  |
| --- |
| The best overloaded method match for 'AnimalShelter< Dog>.Shelter(Dog)' has some invalid arguments Argument 1: cannot convert from 'Cat' to 'Dog' |

**Типизиране на методи**

Подобно на класовете, когато при декларацията на един метод, не можем да кажем от какъв тип ще са параметрите му, можем да типизираме метода. Съответно, указването на конкретния тип ще стане по време на извикване на метода, заменяйки непознатият тип с конкретен, както направихме при класовете.

Типизирането на метод се прави, като веднага след името и преди отва­рящата кръгла скоба на метода, се добави **<K>**, където **K** е заместителят на типа, който ще се използва в последствие:

|  |
| --- |
| **<return\_type> <methods\_name><K>(<params>)** |

Съответно, можем да използваме неизвестния тип **K** за параметрите в списъка с параметри на метода **<params>**, чийто тип не ни е известен, а също и като връщана стойност или за деклариране на променливи от типа заместител **K** в тялото на метода.

Например, нека разгледаме един метод, който разменя стойностите на две променливи:

|  |
| --- |
| public void Swap<K>(ref K a, ref K b){ K oldA = a; a = b; b = oldA;} |

Това е метод, който разменя стойностите на две променливи, без да се интересува от типа им. Затова сме го типизирали, за да можем да го прилагаме за всякакви типове променливи.

Съответно, ако искаме да разменим стойностите на две целочислени и след това на две низови променливи, бихме използвали нашия метод :

|  |
| --- |
| int num1 = 3;int num2 = 5;Console.WriteLine("Before swap: {0} {1}", num1, num2);// Invoking the method with concrete type (int)Swap<int>(ref num1, ref num2);Console.WriteLine("After swap: {0} {1}\n", num1, num2); string str1 = "Hello";string str2 = "There";Console.WriteLine("Before swap: {0} {1}!", str1, str2);// Invoking the method with concrete type (string)Swap<string>(ref str1, ref str2);Console.WriteLine("After swap: {0} {1}!", str1, str2); |

Когато изпълним този код, резултатът ще е както очакваме:

|  |
| --- |
| Before swap: 3 5After swap: 5 3 Before swap: Hello There!After swap: There Hello! |

Забелязваме, че в списъка с параметри сме използвали също и ключовата дума **ref**. Това е така, заради спецификата на това което прави методът – а именно да размени стойностите на две референции. При използването на ключовата дума **ref**, методът ще използва същата референция, която е подадена от извикващия метод. По този начин, всички промени, които са направени от нашия метод върху тази променлива, ще се запазят след приключване работата на нашия метод и връщане на контрола върху изпълнението на програмата обратно на извикващия метод.

Трябва да знаем, че при извикване на типизиран метод, можем да пропуснем изричното деклариране на конкретния тип (в нашия пример **<int>**), тъй като компилаторът ще го установи автоматично, разпозна­вайки типа на подадените параметри. С други думи, нашият код може да бъде опростен използвайки следните извиквания:

|  |
| --- |
| Swap(ref num1, ref num2); // Invoking the method Swap<int>Swap(ref str1, ref str2); // Invoking the method Swap<string> |

Трябва да знаем, че компилаторът ще може да разпознае какъв е конкрет­ния тип, само ако този тип участва в списъка с параметри. Компилаторът не може да разпознае какъв е конкретния тип на типизиран метод само от типа на връщаната стойност на метода или в случай, че методът е без параметри. В тези случаи, конкретния тип ще трябва да бъде подаден изрично. В нашия пример, това ще стане по подобие на първоначалното извикване на метода, чрез добавяне **<int>** или **<string>**.

Трябва да отбележим, че статичните методи също могат да бъдат типизи­рани за разлика от свойствата и конструкторите на класа.

|  |  |
| --- | --- |
| clip_image007[27] | **Статичните методи също могат да бъдат типизирани, дока­то свойства и конструкторите на класа не могат.** |

**Особености при деклариране на типизирани методи в типизирани класове**

Както видяхме в секцията "[Използване на неизвестните типове в декларацията на методи](http://www.introprogramming.info/intro-csharp-book/read-online/glava14-definirane-na-klasove/#_%D0%98%D0%B7%D0%BF%D0%BE%D0%BB%D0%B7%D0%B2%D0%B0%D0%BD%D0%B5_%D0%BD%D0%B0_%D0%BD%D0%B5%D0%B8%D0%B7%D0%B2%D0%B5%D1%81%D1%82%D0%BD%D0%B8%D1%82%D0%B5)", нетипизираните методи могат да из­пол­зват неизвестните типове, описани в декларацията на типизирания клас (например методите **Shelter()** и **Release()** от примера за приюта за бездомни животни):

|  |
| --- |
| **AnimalShelter.cs** |
| public class AnimalShelter<T>{ // ... rest of the code ...  public void Shelter(T newAnimal)  {  // Method body here }  public T Release(int i)  {  // Method body here  }} |

Ако обаче, се опитаме да преизползваме променливата, с която сме озна­чили непознатия тип на типизирания клас, например **T**, при декларацията на типизиран метод, тогава при опит за компилиране на класа, ще полу­чим предупреждение (warning) CS0693, тъй като в областта на действие, на неизвестния тип **T**, дефиниран при декларацията на метода, припокри­ва областта на действие на неизвестния тип **T**, в декларацията на класа:

|  |
| --- |
| **CommonOperations.cs** |
| public class CommonOperations<T>{ // CS0693 public void Swap<T>(ref T a, ref T b) { T oldA = a; a = b; b = oldA; }} |

При опит за компилация на този клас, ще получим следното съобщение:

|  |
| --- |
| Type parameter 'T' has the same name as the type parameter from outer type 'CommonOperations<T>' |

Затова, ако искаме нашият код да е гъвкав, и нашият типизиран метод безпроблемно да бъде извикван с конкретен тип, различен от този на типизирания клас при инстанцирането на класа, просто трябва да декларираме заместителя на неизвестния тип в декларацията на типизи­рания метод, да бъде различен от параметъра за неизвестния тип в декла­рацията на класа, както е показано по-долу:

|  |
| --- |
| **CommonOperations.cs** |
| public class CommonOperations<T>{ // No warning public void Swap<K>(ref K a, ref K b) { K oldA = a; a = b; b = oldA; }} |

По този начин, винаги ще сме сигурни, че няма да има препокриване на заместителите на неизвестните типове на метода и класа.

**Използването на ключовата дума default в типизиран код**

След като се запознахме с основите на типизирането, нека се опитаме да преработим нашия пръв пример в тази секция – [класът описващ приют за бездомни животни](http://www.introprogramming.info/intro-csharp-book/read-online/glava14-definirane-na-klasove/#_%D0%9F%D1%80%D0%B8%D1%8E%D1%82_%D0%B7%D0%B0_%D0%B1%D0%B5%D0%B7%D0%B4%D0%BE%D0%BC%D0%BD%D0%B8). Както разбрахме, единственото, което е нужно да направим е да заменим конкретния тип **Dog** с някакъв параметър, например **T**:

|  |
| --- |
| **AnimalsShelter.cs** |
| public class AnimalShelter<T>{ private const int DefaultPlacesCount = 20;  private T[] animalList; private int usedPlaces;  public AnimalShelter() : this(DefaultPlacesCount) { }  public AnimalShelter(int placesCount) { this.animalList = new T[placesCount]; this.usedPlaces = 0; }  public void Shelter(T newAnimal) { if (this.usedPlaces >= this.animalList.Length) { throw new InvalidOperationException("Shelter is full."); } this.animalList[this.usedPlaces] = newAnimal; this.usedPlaces++; }  public T Release(int index) { if (index < 0 || index >= this.usedPlaces) { throw new ArgumentOutOfRangeException( "Invalid cell index: " + index); } T releasedAnimal = this.animalList[index]; for (int i = index; i < this.usedPlaces - 1; i++) { this.animalList[i] = this.animalList[i + 1]; } this.animalList[this.usedPlaces - 1] = null; this.usedPlaces--;  return releasedAnimal; }} |

Всичко изглежда наред, докато не се опитаме да компилираме класа. Тогава получаваме следната грешка:

|  |
| --- |
| Cannot convert null to type parameter 'T' because it could be a non-nullable value type. Consider using 'default(T)' instead. |

Грешката е в метода **Release()** и е свързана със записването на резултат **null** в освободената последна (най-дясна) клетка на приюта. Проблемът е, че се опитваме да използваме подразбиращата се стойност за референ­тен тип, но не сме сигурни, дали конкретния тип е референтен или примитивен. Тъкмо затова, компилаторът извежда гореописаните грешки. Ако типът **AnimalShelter** се инстанцира по структура, а не по клас, то стойността null е невалидна.

За да се справим с този проблем, трябва в нашия код, вместо **null**, да използваме конструкцията **default(T)**, която връща подразбиращата се стойност за конкретния тип, който ще бъде използван на мястото на **T**. Както знаем подразбиращата стойност за референтен тип е **null**, а за числови типове – нула. Можем да направим следната промяна:

|  |
| --- |
| // this.animalList[this.usedPlaces - 1] = null;this.animalList[this.usedPlaces - 1] = default(T); |

Едва сега компилацията минава без проблем и класът **AnimalShelter<Т>** работи коректно. Можем да го тестваме например по следния начин:

|  |
| --- |
| static void Main(){ AnimalShelter<Dog> shelter = new AnimalShelter<Dog>(); shelter.Shelter(new Dog()); shelter.Shelter(new Dog()); shelter.Shelter(new Dog()); Dog d = shelter.Release(1); // Release the second dog Console.WriteLine(d); d = shelter.Release(0); // Release the first dog Console.WriteLine(d); d = shelter.Release(0); // Release the third dog Console.WriteLine(d); d = shelter.Release(0); // Exception: invalid cell index} |

**Предимства и недостатъци на типизирането**

Типизирането на класове и методи води до по-голяма преизползваемост на кода, по-голяма сигурност и по-голяма ефективност, в сравнение с алтернативните нетипизирани решения.

Като генерално правило, програмистът трябва да се стреми към типизи­ране на класовете, които създава винаги, когато е възможно. Колкото повече се използва типизиране, толкова повече нивото на абстракция в програмата се покачва, както и самият код става по-гъвкав и преизпол­зваем. Все пак трябва да имаме предвид, че прекалената употреба на типизиране може да доведе до прекалено генерализиране и кодът може да стане нечетим и труден за разбиране от други програмисти.

**Ръководни принципи при именуването на заместителите при типизиране на класове и методи**

Преди да приключим с темата за типизирането, нека дадем някои указания при работата със заместителите (параметрите) на непознатите типове в един типизиран клас:

1.  Когато при типизирането имаме само един непознат тип, тогава е общоприето да се използва буквата **T**, като заместител за този непознат тип. Като пример можем да вземем декларацията на нашия клас **AnimalShelter<T>**, който използвахме до сега.

2.  На заместителите трябва да се дават възможно най-описателните имена, освен ако една буква не е достатъчно описателна и добре подбрано име, не би подобрило по никакъв начин четимостта на кода. Например, можем да модифицираме нашия пример, заменяйки буквата **T**, с по-описателния заместител **Animal**:

|  |
| --- |
| **AnimalShelter.cs** |
| public class AnimalShelter<Animal>{ // ... rest of the code ...  public void Shelter(Animal newAnimal)  {  // Method body here }  public Animal Release(int i)  {  // Method body here  }} |

Когато използваме описателни имена на заместителите, вместо буква, е добре да добавяме **T**, в началото на името, за да го разграничаваме по-лесно от имената на класовете в нашата програма. С други думи, вместо в предходния пример да използваме заместител **Animal**, е добре да използваме **TAnimal**.