Класове и обекти - II

Конструктори

Конструкторът е метод, които няма тип на връщана стойност, носи името на класа и който се извиква чрез ключовата дума **new**. Той се използва при създаване на обекти. Задачата му е да **инициализира** заделената за обекта памет, в която ще се съхраняват неговите полетата (тези, които не са static).
Инициализациите, описани в тялото на конструктора се изпълняват по време на изпълнението този конструктор – при създаване на обект от съответния клас с ключовата дума **new** в C#.
Инициализациите, дефинирани при декларацията на полетата се изпълняват директно преди конструктора.
Механизмът на работа и синтаксисът за дефиниране на конструкторите в C# са подобни на други обектно-ориентирани езици, като Java и C++, с някои малки особености.

Допуска се използването на повече от един конструктор, като те трябва да се различават по броя и/или типа на параметрите си. Възможно е и да не се дефинира конструктор и в такъв случай компилаторът създава подразбиращ се – публичен, с празно тяло и без параметри.

Пример

```
public  class Student
{
        private string mName;
        private int mStudentId;
        private string mPosition = "Student";
        // следва конструктор с 2 аргумента
        public Student (string aName, int aStudentId)
        {
                mName = aName;
                mStudentId = aStudentId;
        }

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

Индексатори

Индексаторите в C# (indexers) са членове на класовете, структурите и интерфейсите, които предоставят индексиран достъп до данни на типа, подобно на достъпа до елементите на масив.
Индексаторите по синтаксис и семантика много приличат на свойства, но получават като параметър индекс на елемент, с който да работят. На практика, те представляват свойства, приемащи параметър и дори в някои .NET езици, например VB.NET, синтаксисът на деклариране им е същият като при свойствата.

Пример

```
private object[] mElements;
public object this[int index]
{
        get{return mElements[index];}
}
```

На индексатора не се задава име, а вместо него се задава запазената дума **this**

Достъп до индексатор на обект се извършва посредством името на променливата от типа, дефиниращ индексатора, последвана от индекса в квадратни скоби, също както достъп до елемент на масив. Индексаторите може да се разглеждат като средство за предефиниране на оператора **[ ]**. Използването на индексатори позволява интуитивен достъп до обекти, които се състоят от множество компоненти, каквито са масивите и колекциите.

Интерфейси

Интерфейсите описват функционалност (група методи, свойства, индексатори и събития), която се поддържа от множество обекти. Подобно на класовете и структурите те се състоят от членове, но се различават от тях по това, че дефинират само прототипите на членовете си, без конкретната им реализацията.

От интерфейсите не могат да се създават обекти чрез инстанциране.

Интерфейсите се **реализират** от класове или структури, които имплементират всички дефинирани в тях членове.

Конкретните имплементации на даден интерфейс вече могат да се инстанцират и да се присвояват на променливи от тип интерфейс.

Членове на интерфейс

Интерфейсите могат да съдържат методи, свойства, индексатори и събития. В интерфейс не могат да се дефинират конструктори, деструктори, полета и вложени типове и не могат да се предефинират оператори.

Интерфейсите в C# не могат и да съдържат и константи, за разлика от други обектно-ориентирани езици, като Java, където това е допустимо.

Към членовете на интерфейс не може да се прилагат модификатори на достъпа – по подразбиране всички членове са с глобална видимост, все едно е указан модификатор **public**. Интерфейс може да наследи един или повече интерфейси, като е възможно да предефинира или скрива техните членове.

Пример

```
interface Ishape
{           void SetPosition(int aX, int aY);
            double CalculateSurface();
}

interface IPerimeterShape : IShape
{           double CalculatePerimeter();
}

interface IResizable
{           void Resize(int aWeight);
            void Resize(int aWeightX, int aWeightY);
            void ResizeByX(int aWeightX);
            void ResizeByY(int aWeightY);
}

interface IDrawableShape : IShape, IResizable
{           void Delete();
            Color Color
            {           get;
                        set;
            }
}
```

Дефинирани са интерфейси: IShape, IPerimeterShape, IResizable и IDrawableShape. Те илюстрират дефинирането на методи и свойства в интерфейс, както и наследяването между интерфейси (което може да бъде и множествено, например при IDrawableShape).

Реализиране на интерфейс

Тъй като не съдържат данни и описана функционалност, интерфейсите не могат да се инстанцират, а само да се **реализират** (**имплементират**) от класове и структури, от които вече могат да се създават инстанции.
Реализирането на интерфейс е операция, подобна на наследяването, с тази особеност, че реализиращият интерфейса тип в общия случай трябва да предостави реализации за всички членове на интерфейса.

Пример

```
public class Square : Ishape
{       private int mX, mY, mSize;
        public Square(int aX, int aY, int aSize)
        {       mX = aX; mY = aY; mSize = aSize;
        }
        public void SetPosition(int aX, int aY) // от IShape
        {       mX = aX; mY = aY;
        }
        public double CalculateSurface() // наследено от IShape
        {       return mSize * mSize;
        }
}
```

Класът **Square** реализира интерфейса **IShape**

Пример

```
public class Circle : IPerimeterShape
{       private int mX, mY, mRadius;
        public Circle(int aX, int aY, int aRadius)
        { mX = aX; mY = aY; mRadius = aRadius;
        }
        public void SetPosition(int aX, int aY) // от IShape
        {mX = aX; mY = aY;
        }
        public double CalculateSurface() // от IShape
        { return Math.PI * mRadius * mRadius;
        }
        public double CalculatePerimeter() // от IPerimeterShape
        {return 2 * Math.PI * mRadius;
        }
}
```

Класът Circle реализира интерфейса IPerimeterShape, но понеже този интерфейс е наследник на IShape, това означава, че Circle на практика имплементира едновременно интерфейсите IShape и IPerimeterShape. Забележете, че всички методи в даден интерфейс са публични (въпреки, че нямат модификатор public).

Обекти от тип интерфейс

С пример ще демонстрираме създаването на обекти от тип интерфейс. Реално ще създаваме обекти от типове (класове), които наследяват даден интерфейс, по-точно през променлива от тип интерфейс ще достъпваме обекти от класовете, които реализират (имплементират) този интерфейс.

Пример

```
class GeomertyTest
{       public static void Main()
        {       Square square = new Square(0, 0, 10); //обект квадрат
                Circle circle = new Circle(0, 0, 5); //обект кръг
                if (square is IShape)
                {Console.WriteLine("{0} is IShape", square.GetType());}
                IShape[] shapes = {square, rect, circle};
                foreach (IShape shape in shapes)
                {       shape.SetPosition(5, 5);
                        if (shape is IPerimeterShape)
                        {Console.WriteLine("{0} is IPerimeterShape", shape);}
                }
        }
}
```

В примера се създава масив от обекти от тип Ishape, към които се прилага действието SetPosition(…) полиморфно, т. е. без да се интересуваме от типа им – единствено знаем, че обектите поддържат методите от интерфейса.

Явна имплементация на интерфейс

Класовете и структурите могат да имплементират по повече от един интерфейс (множествено наследяване). Това би могло да създаде конфликт, ако един тип имплементира няколко интерфейса, съдържащи методи с еднакви сигнатури. Да разгледаме следния пример:

```
public interface InterFace1
{   void Test();}
public interface InterFace2
{   void Test();
     void AnotherTest();}
public class TestImplementation : InterFace1, InterFace2
{           public void Test()
            {Console.WriteLine("Test() called");}
}
```

Този код е допустим в C#, но не се препоръчва, защото създава затруднения.

За да се справим с описания проблем можем да използваме **явната имплементация на интерфейси** (explicit interface implementation).

Явна имплементация на интерфейс

В C# можем да дефинираме в един тип два метода с еднаква сигнатура, стига поне единият от тях да е явна имплементация на метод от интерфейс. Явна имплементация се задава, като изрично се укаже на кой интерфейс принадлежи имплементираният член.

```
public class TestExplicit : InterFace1, InterFace2
{           void InterFace1.Test()
            {Console.WriteLine("InterFace1.Test() called");}
            void InterFace2.Test()
            {Console.WriteLine(" InterFace2.Test called");}
            void InterFace2.AnotherTest()
            {Console.WriteLine("InterFace2.AnotherTest called");}
            public void Test()
            {Console.WriteLine("TestExplicit.Test() called");}

            public static void Main()
            {           TestExplicit t = new TestExplicit();
                        t.Test();      // Отпечатва: TestExplicit.Test() called
                         InterFace1 i1 = (InterFace1) t;
                        i1.Test();    // Отпечатва: InterFace1.Test() called
                         InterFace2 i2 = (InterFace2) t;
                        i2.Test();     // Отпечатва: InterFace2.Test() called  }
}
```

При явна имплементация на интерфейс трябва да укажем името му в дефиницията на реализирания член, а за да го достъпим трябва да **преобразуваме обекта към интерфейса**. Методите, принадлежащи на явно имплементиран интерфейс са винаги **private**.

Полиморфизъм (Polymorphism)

Думата полиморфизъм буквално означава „многообразие”. В абстрактен ред на мисли тя обозначава свойството на някаква инстанция да притежава много форми, чрез които може да се проявява. Понятието „полиморфизъм“ идва от биологията. То описва „съществуване на морфологично различни индивиди в границите на един вид“. Това например са пчелите и търтеите в пчелния кошер – те са от един вид, но имат съвсем различни функции. В обектно-ориентираното програмиране под „полиморфизъм“ се разбира същото – различни обекти (инстанции) от един и същи вид (базов клас) извършват различни дейности. Тук това обаче трябва да се разбира в смисъл, че инстанциите-обекти **наследяват** характерните черти на вида (базовия клас), **но променят** някои от неговите функционалности.

Полиморфизмът може много да напомня на абстракцията, но в програмирането се свързва най-вече с пренаписването (**override**) на методи в наследените класове с цел промяна на оригиналното им поведение, наследено от базовия клас. Абстракцията се свързва със създаването на интерфейс на компонент или функционалност (дефиниране на роля).

Абстрактни класове

Абстрактните класове приличат на интерфейсите по това, че не могат да се инстанцират (да имат обекти), защото могат да съдържат дефиниции на неимплементирани методи, но за разлика от интерфейсите могат да съдържат и описани действия. Абстрактният клас реално е комбинация между клас и интерфейс – частично имплементиран клас, който дефинира имплементация за някои от методите си, а други оставя абстрактни, без имплементация.

Пример

```
public abstract class Car
{
    public void Move()
        {// Move the car}
    abstract public int TopSpeed
        {  get;}
     public abstract string BrandName
        {  get;}
}
```

Дефиниран е клас, който реализира само един от членовете си – метода **Move()** и дефинира други два, без да ги реализира – свойствата **TopSpeed** и **BrandName**.

Абстрактните класове, подобно на интерфейсите, по-адекватно моделират зависимости от реалният свят, защото чрез тях могат да се представят абстрактни същности. В примера, невъзможността за инстанциране на класа **Car** има смисъл, тъй като и в реалността не можем да имаме кола с неопределена марка

Абстрактни членове

Ключовата дума **abstract** в декларацията на класа го определя като абстрактен. В примера се вижда, че тя може да се приложи и към член. Абстрактни могат да бъдат методите, свойствата, индексаторите и събитията.
Ако в един клас е дефиниран абстрактен член, класът задължително трябва да бъде обявен за абстрактен. В противен случай получаваме грешка при компилация. Обратното не е задължително – допустимо е да имаме абстрактен клас, на който всички членове са дефинирани.
Абстрактните класове са нещо средно между клас и интерфейс. Те могат да дефинират обикновени методи и абстрактни методи. Обикновените методи имат тяло (имплементация), докато абстрактните методи са празни (без имплементация) и са оставени да бъдат реализирани от класовете-наследници.

Абстрактните членове не могат да имат имплементация, както и член, който не е абстрактен, не може да бъде оставен без такава.

Наследяване на абстрактни класове

Тъй като абстрактните класове са класове, те имат същата структура - същият набор от членове (полета, константи, вложени типове и т. н.), същите модификатори на видимостта и дори същите механизми за наследяване, но с някои особености.

В следващия пример се демонстрира как въпреки, че абстрактен клас не може да се инстанцира директно, обектите от наследяващите го класове могат да се разглеждат като обекти от неговия тип. По показания начин можем да използваме абстрактни базови класове, за да задействаме полиморфизъм, или, казано по-общо, да създадем абстрактен корен на дърво от класове.

В примера се ползва ключовата дума **override**, с която се указва, че даден метод в класа наследник припокрива (замества) оригиналния наследен метод от базовия си клас. В случая базовия клас не предоставя имплементация за припокритите методи, така че припокриването е задължително.

Забележка: Възможно е абстрактен клас да бъде наследен без всичките му абстрактни членове да бъдат реализирани. Възможно е също клас, който имплементира абстрактните членове на абстрактния си родител, да дефинира допълнително и свои абстрактни членове. В такива случаи класът-наследник също трябва да бъде деклариран като абстрактен.

Пример

```
public class Trabant : Car
{           public override int TopSpeed
            {   get {return 120;}
            }
            public override string BrandName
            {    get {return "Trabant";}
            }
}
public class Porsche : Car
{           public override int TopSpeed
            {    get {return 250;}
            }
            public override string BrandName
            {     get {return "Porsche";}
            }
}
public class AbstractTest
{ static void Main()
  {Car[] cars = new Car[] {new Trabant(), new Porsche()};
  foreach (Car car in cars)
  {Console.WriteLine("A {0} can go {1} Kmph", car.BrandName, car.TopSpeed);}
 }
}
```

Обекти – инстанции на класове

Софтуерните обекти моделират обекти от реалния свят или абстрактни концепции (които също разглеждаме като обекти).

Всеки конкретен обект е представител на точно един клас и е създаден по шаблона на този клас. Създаването на обект от вече дефиниран клас наричаме **инстанциране** (instantiation), а самият обект - **инстанция** на този клас.

Всяка инстанция се характеризира от две неща: нейното моментно състояние (**state**) и нейното поведение (**behavior**).

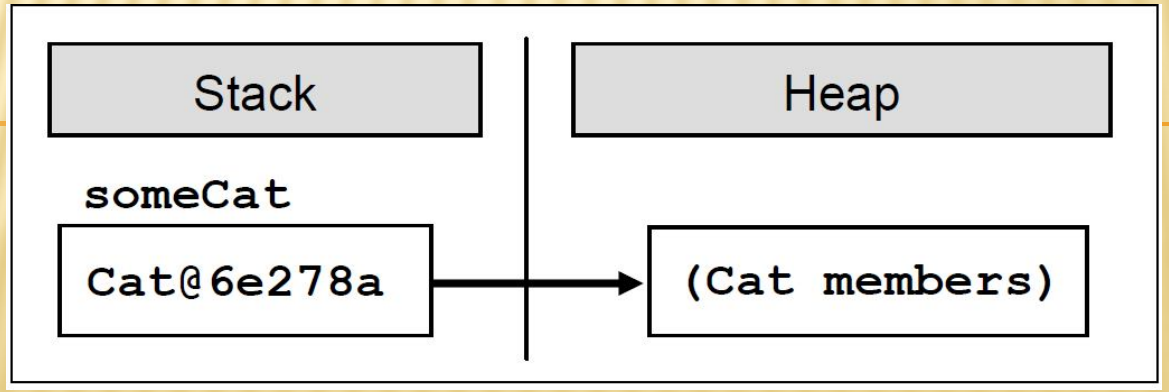
- Моментното състояние на инстанция се представлява от множество от стойности, асоциирани с атрибутите на класа и е специфично за конкретната инстанция;
- Поведението на инстанция е дефинирано в класа й и е общо за всички инстанции (обекти), които са представители на този клас.

Създаване и освобождаване на обекти

Създаването на обекти от предварително дефинирани класове по време на изпълнението на програмата става чрез оператора **new**. Новосъздаденият обект обикновено се присвоява на променлива от тип, съвпадащ с класа на обекта (това обаче не е задължително). Ще отбележим, че при това присвояване същинският обект не се копира, а в променливата се записва само референция към новосъздадения обект (неговият адрес в паметта). Следва прост пример как става това:

```
Cat someCat = new Cat();
```

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



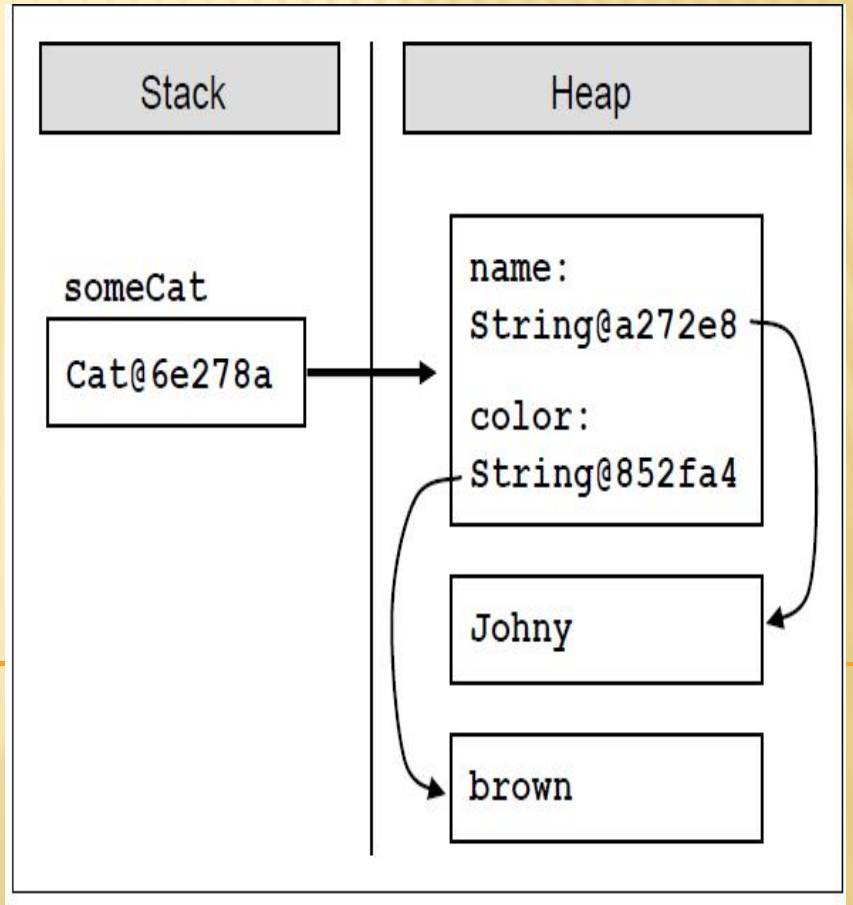
Създаване на обекти със задаване на параметри

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

```
Cat someCat = new Cat("Johnny", "brown");
```

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

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



Освобождаване на обектите

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

```
someCat = null;
```

Достъп до полета на обекта

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

В случай на свойство, достъпът се реализира по абсолютно същия начин както и при поле – C# ни предоставя тази възможност. Това се постига чрез двете специални ключови думи get и set в дефиницията на свойството, които извършват съответно извличането на стойността на свойството и присвояването на нова стойност. В дефиницията на класа Cat, която даваме в следващия пример, свойства са Name и Color.

Пример

```
public class Cat
{   private string name;
    private string color;
    public string Name
   { get {return this.name;}
     set {this.name = value;}
    }
    public string Color
    { get {return this.color;}
      set {this.color = value;}
    }
    public Cat () // Default constructor
    { this.name = "Unnamed";
      this.color = "gray";
    }
    public Cat (string name, string color)   // Constructor
    { this.name = name;
       this.color = color;
    }
    public void SayMiau()
    {Console.WriteLine("Cat {0} said: Miauuuuuu!", name);}
}
```