Обектно ориентирано програмиране (C#)

Лекция №4 Стойности и референции.

Всички типове в C# попадат в една от следните категории:

- стойностни (*value types*);
- референтни (*reference types*);
- параметри на генерични типове (*generic type parameters*);
- указателни (*pointer types*).

Днес ще разгледаме първите две категории.

➢ **Стойностни и референтни типове**

Повечето примитивни типове в C# като `int`, `float`, `double`, `bool` и `char` (но не и `string`) се наричат стойностни типове (*value types*). Към тях спадат още дефинираните в програмите структури (`struct`) и изброявания (`enum`). Тези типове имат фиксиран размер и когато декларираме променлива от такъв тип, компилаторът заделя парче памет, достатъчно голямо, за да може да съхрани стойност от този тип. Например за променлива `i` от тип `int` компилаторът ще задели 4 байта (32 бита). Оператор, присвояващ стойност 42 на тази променлива ще предизвика копиране на стойността в заделеното парче памет:

```
int i;
i = 42;
```

Класове, като разгледания в предишната лекция `Circle` се обработват различно. Когато декларираме променлива от тип `Circle`, компилаторът не генерира код, който заделя блок от памет, достатъчно голям да съхрани обект на `Circle`. Вместо това той заделя малко парче памет, което потенциално може да съхрани адрес (или референция) на друго парче памет, съдържащо обект на `Circle`. Памет за действителен обект на `Circle` се заделя само когато се използва ключовата дума `new` . Класът е пример за референтен тип (*reference type*). Такива типове съдържат референции към блокове памет. Трябва добре да се разбира разликата между двата вида типове.

Типът `string` всъщност е клас, а ключовата дума `string` е псевдоним на класа `System.String`. Към референтните типове спадат още масиви, делегати и интерфейси.

Да разгледаме следния пример:

```
int i = 42;
int copyi= i;
i++;
Console.WriteLine($"i = {i}, copyi = {copyi}");
```

На екрана ще се изпише

```
i = 43, copyi = 42
```

защото за двете променливи са заделени два различи блока памет. Увеличаването на стойността на `i` не влияе на `copyi`.

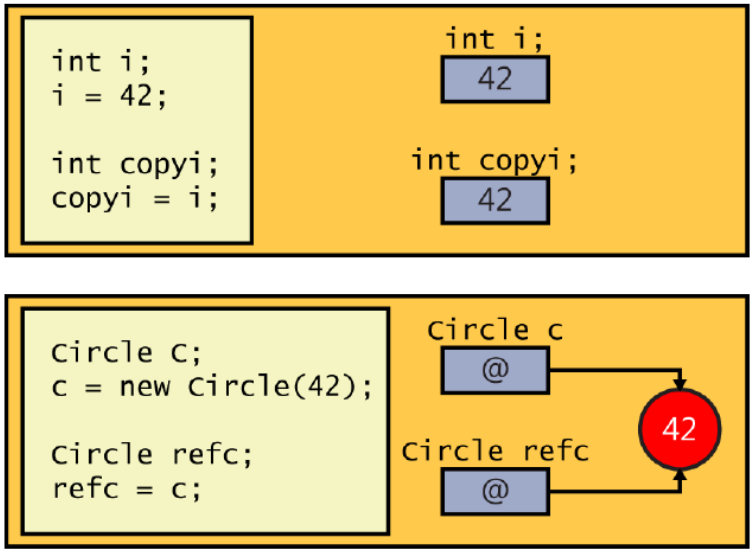
Ситуацията е коренно различна, ако вместо с типа `int` работим с обекти на класа `Circle`:

```
Circle c = new Circle(42);
Circle refc = c;
refc.SetRadius(10);
Console.WriteLine($"Area of c: {c.Area()}, Area of refc: {refc.Area()}");
```

Ефектът от декларирането на променливата `c` от тип клас, какъвто е `Circle` е много различен. Декларацията само казва, че `c` може да сочи (да бъде референция) към обекти на `Circle`. Стойността, съхранявана в `c` е адрес на обект на `Circle` в паметта. Ако декларираме друга

променлива (`refc`) и ѝ присвоим стойност `c`, в нея ще имаме копие на същия адрес като този, съхраняван в `c`. Има само един обект на `Circle` – този, създаден с оператора `new` и двете променливи съдържат референции към него. Затова промяната на радиуса чрез `refc` се отразява и на `c`.

Двете ситуации са показани на Фигура 1.



Фиг. 1 Стойностни и референтни типове

Копиране на референтни типове

Ако искаме да имаме два еднакви обекта, вместо две референции към един и същ обект, трябва да се погрижим за това, като дефинираме метод на класа, който връща резултат нов обект, съдържащ същите данни. Такъв метод обикновено носи името `Clone()`. Като изучаваме интерфейсите ще стане въпрос за този метод и интерфейса `ICloneable`.

За класа `Circle` този метод изглежда така:

```
public Circle Clone() => new Circle(this.radius);
```

Ето пример за използването му:

```
Circle copyc = c.Clone();
Console.WriteLine($"Area of c = {c.Area()}, Area of copyc = {copyc.Area()}");
copyc.SetRadius(50);
Console.WriteLine($"Area of c = {c.Area()}, Area of copyc = {copyc.Area()}");
```

Същият ефект можем да получим, ако дефинираме конструктор с параметър от тип същия клас:

```
public Circle (Circle other) // нещо като копиращ конструктор
{
    this.radius = other.radius;
}
```

И да го използваме така:

```
Circle copyc = new Circle(c);
Console.WriteLine($"Area of c = {c.Area()}, Area of copyc = {copyc.Area()}");
copyc.SetRadius(50);
Console.WriteLine($"Area of c = {c.Area()}, Area of copyc = {copyc.Area()}");
```

Разликата е, че конструкторът се извиква само при създаването на обект, а методът `Clone()` може да се използва за присвояване на стойност на вече създаден обект (по-точно казано – за пренасочване на референцията към друг обект).

Ще обърнем внимание на ситуацията, когато референтен тип се предава като аргумент на функция по стойност. Прави се копие на референцията, но това копие съдържа същия адрес на сочения от оригиналната референция обект, така че обектът е достъпен и може да бъде модифициран. Ако въпросната функция не е метод на класа тя има достъп само до `public` членовете му, тоест класът си остава капсулиран.

Пример:

```
static void ResizeCircle(Circle c, int radius)
{
    c.SetRadius(radius);
}
```

И после в `Main()`

```
ResizeCircle(c, 1);
Console.WriteLine(c.Area());
```

Ако обаче работим с целите обекти, например искаме да дефинираме функция, която разменя две окръжности, трябва да предадем параметрите като референции (`ref`), ако искаме размяната да се извърши:

```
public static void Swap(ref Circle a, ref Circle b)
{
    Circle t = a;
    a = b;
    b = t;
}
```

Съответно в `Main()`:

```
Circle c1 = new Circle(1),
       c2 = new Circle(10);
Swap(ref c1, ref c2);
Console.WriteLine($"c1: {c1.Area()}");
Console.WriteLine($"c2: {c2.Area()}");
```

Пробвайте какво ще е поведението на програмата без `ref` в параметрите и аргументите.

- **Стойност `null` и *nullable* типове**

Когато декларираме променлива е добре да я инициализираме още в рамките на декларацията. За стойностните типове това се прави така:

```
int i = 0;
double d = 0.0;
```

Инициализирането на променлива от референтен тип изисква да ѝ се присвои стойност на инстанция (обект) на съответния тип (клас) – или вече съществуващ, или създаден с оператора `new` в рамките на декларацията:

```
Circle c = new Circle(10);
```

Ако обаче не искаме да създаваме обект, а да ползваме променливата само като референция на обект, който ще бъде създаден по-късно? Оправдано ли е да я инициализираме с някакъв случаен обект, само за да не остане неинициализирана (виж примера)?

```
Circle c = new Circle(10);
Circle copy = new Circle(99);
//.......................
copy = c;
//.............................
```

Какво се случва с обекта на `Circle` с радиус 99, след като `copy` вече сочи към `c`? Към този обект вече не сочи никоя референция и той е недостъпен за програмата. В този случай средата (*runtime*) може да рециклира паметта, извършвайки операция, известна като „събиране на боклука“ (*garbage collection*). Това обаче е потенциално „скъпа“ операция по отношение на време и затова не е оправдано да създаваме обекти, които не използваме. Оставянето на променливата неинициализирана е едно решение, но лошо решение, което може да доведе до проблеми, например в ситуация, ако искаме на дадена променлива да присвоим нова стойност, само ако вече не сочи към друг обект:

```
Circle c = new Circle(10);
Circle copy;    // неинициализирана
// ...
if (copy == // искам да проверя дали е неинициализирана, но как?)
{
    copy = c;
    //.............................
}
```

Въпросът е с коя стойност да сравня `copy`, за да проверя дали е инициализирана или не? За тази цел е предвидена специална стойност, означена с ключовата дума `null`. Тази стойност може да се присвоява на променливи от всички референтни типове. Горният пример вече изглежда така:

```
Circle c = new Circle(10);
Circle copy = null;    // вече инициализирана
// ...
if (copy == null )
{
    copy = c;
    //.............................
}
```

Използването на променлива, която сочи `null`, като такава, която сочи действителен обект води до грешка по време на изпълнение на програмата:

```
Circle c = new Circle(10);
Circle copy = null;
copy.SetRadius(15);    // System.NullReferenceException
```

Nullable типове

Стойността `null` по същество е референция и не може да се присвоява на променливи от *value* типове.

```
string s = null;    // OK, Reference Type
int i = null;       // Compile Error, Value Type cannot be null
```

Затова в C# е предвидена конструкция, наречена *nullable value type*. Такъв тип се получава като към името на обикновен стойностен тип добавим въпросителен знак (?). Променлива от този тип се държи по същия начин като от обикновения тип, но можем да ѝ присвояваме стойност `null`.

```
int? i = null;       // OK
```

Можем да проверим дали променлива от *nullable* тип съдържа `null` по същия начин, както правим това с референтните типове:

```
if(i == null)
    Console.WriteLine("i is null");
else
    Console.WriteLine($"i= {i}");
```

Можем на *nullable* променлива директно да присвоим израз от съвместим *value* тип:

```
int? i = null;
int j = 99;
i = 100;  // Copy a value type constant to a nullable type
i = j-89; // Assign a value type expression to a nullable type
Console.WriteLine("i = {0}",i); // 10
```

Не е допустимо обаче на променлива от стойностен тип директно (без явно преобразуване) да се присвои *nullable* стойност:

```
j = i;
```

Възниква грешка при компилиране

```
error CS0266: Cannot implicitly convert type 'int?' to 'int'…
```

Това важи и за предаване на *nullable* аргумент като параметър на функция (метод), която очаква обикновен *value* тип. Например не можем да извикаме функцията

```
static int sum(int a, int b) => a + b;
```

с аргументи `i` и `j`:

```
int s = sum(i, j);
```

Nullable типовете предоставят двойка свойства (*properties*), с които можем да проверим дали съдържащата се стойност е различна от `null` и колко е тази стойност. Това са съответно свойствата `HasValue` и `Value`.

```
if (i.HasValue)
    Console.WriteLine("i = {0}", i.Value);
else
    i = 99;
```

Такава проверка и извличане на стойност чрез свойство изглежда по-тежко и претрупано от директно сравняване с `null`, но това е така, защото използваме съвсем прост *value* тип – `int`.

```
if (i != null)
    Console.WriteLine("i = {0}", i);
else
    i = 99;
```

Ще отбележим, че и двете свойства са *read-only*.

Сега ще направим малко пояснение за реализацията на *nullable* типовете, като ще разберете за какво става въпрос, когато изучим в детайли генеричните типове...

Структурата `Nullable<T>`

`T?` се преобразува в `System.Nullable<T>`. Именно в нея са дефинирани двете свойства `HasValue` и `Value`, както и някои предефинирани методи, които ще пропуснем.

```
public struct Nullable<T> where T : struct
{
    public Nullable(T value);
    public bool HasValue { get; }
```

```
        public T Value { get; }
        public static implicit operator T? (T value);
        public static explicit operator T(T? value);
      . . .
    }
```

И така кодът

```
int? i = null;
Console.WriteLine(i == null); // True
```

се преобразува до

```
Nullable<int> i = new Nullable<int>();
Console.WriteLine(!i.HasValue); // True
```

Опитът да се извлече `Value`, когато `HasValue` е `false` предизвиква изключение.

- **Организация на компютърната памет**

Паметта се използва за съхранение на изпълняваните програми и данните, които те ползват. За да разберем по-добре разликата между *value* и *reference* типовете е полезно да разберем как са организирани данните в паметта.

Операционните системи и средите за изпълнение (*language runtimes*), като тази използвана от C#, често разделят паметта, предназначена за съхранение на данните, на две отделени области с различна организация и начин на управление. Те се наричат традиционно *stack* и *heap*.

Когато извикваме метод, паметта, необходима за съхранение на параметрите му и локалните променливи, винаги се заделя в стека. При завършване на метода (с `return` или изключение), тази памет се освобождава и връща на стека за повторно използване, когато се извика друг метод. Променливите, съхранявани в стека имат добре дефинирана продължителност на живота – те се създават при стартиране на метода и изчезват скоро след неговото приключване. Ще припомним, че в стека се съхраняват и локални променливи, дефинирани в блокове `{. . .}`.

При създаване на обект (инстанция на клас) с ключовата дума `new`, необхходимата памет винаги се заделя в *heap*-а. Видяхме, че един и същ обект може да бъде референциран от няколко места чрез различни променливи. Когато изчезне и последната референция към обекта, заеманата от него памет става налична за повторно използване (макар, че това може да не стане веднага). Така обектите в *heap*-а имат по-недетерминиран жизнен път.

Всички стойностни типове се създават в стека. Всички референтни типове се създават в *heap*-а, макар самите референции (адресите) да са в стека. *Nullable* типовете всъщност са референтни типове и се създават в *heap*-а.

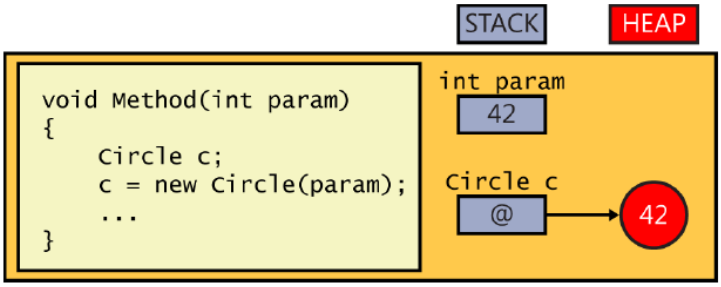
Имената *stack* и *heap* идват от начина на управление на паметта:

- при стека е като камара от кутии, разположени една върху друга. Когато се извика метод, всеки параметър и локална променлива се поставя в кутия на върха на стека. При завършване на метода тези кутии се премахват от стека;
- при *heap*-а камарата с кутии е разпиляна по пода, вместо да са наредени една над друга. Всяка кутия има етикет, показващ дали се използва. Когато се създава нов обект се търси празна кутия и такава се заделя за обекта. Референция към обекта се съхранява в локална променлива в стека. Средата следи броя референции към всяка кутия. Когато и последната референция изчезне, кутията се маркира като неизползвана и по някое време тя ще бъде изпразнена и обявена за налична за повторна употреба.

Да разгледаме един пример с класа `Circle`. Какво се случва при извикването на следния метод:

```
void Method(int param)
{
    Circle c;
    c = new Circle(param);
    //...
}
```

Приемаме, че към `param` е предаден аргумент със стойност 42. Когато методът е извикан, блок от памет (достатъчен за `int`) се заделя в стека и се инициализира с 42. Когато изпълнението започне (контролът е в тялото на функцията) друг блок, достатъчно голям да съхрани референция (адрес) се заделя в стека, но остава неинициализиран. Той е за променливата `c`. След това друго парче памет, достатъчно голямо за обект на `Circle` се заделя в *heap*-а. Това е работата на `new`. Изпълнява се конструкторът на `Circle` и преобразува това сурово парче памет в обект на `Circle`. В променливата `c` се съхранява референция към този обект (фиг. 2).



Фиг. 2. Състояние на паметта при извикване на метод

Ще обърнем внимание на следното:

- макар, че обектът се съхранява в *heap* паметта, референцията към него (променливата `c`) се съхранява в стека;
- *heap* паметта не е безкрайна. Ако тя се изчерпи, операторът `new` ще хвърли изключение *OutOfMemoryException* и обектът няма да бъде създаден;
- конструктор също може да предизвика изключение. Ако това стане, паметта, заделена за обекта ще бъде възстановена и стойността, върната от конструктора ще бъде `null`.

Когато методът завърши работата си параметрите и локалните променливи „излизат от обсег“ (*go out of scope*). Паметта, заделена за `c` и `param` автоматично се освобождава за стека. Средата (*runtime*) отбелязва, че обектът на `Circle` вече не се референсира и в някакъв момент ще възстанови паметта му за *heap*-а.

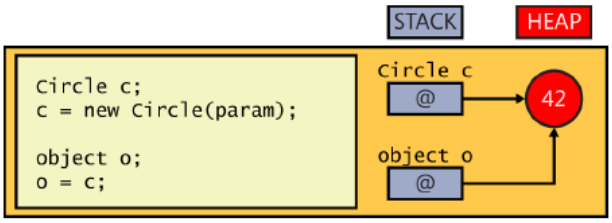
➢ Класът `System.Object`

Може би най-важният референтен тип в *.NET Framework* е класът `Object` в пространството на имената `System`. За да оценим напълно неговата важност трябва да познаваме механизма на наследяване, който ще изучим по-късно. Засега можем да приемем, че всички класове са специализации на `System.Object` и променливи от този тип могат да бъдат референции на обекти на всеки друг клас. Тъй като `System.Object` е толкова важен, C# предоставя ключовата дума `object` като негов псевдоним.

Да разгледаме следния пример:

```
Circle c = new Circle(42);
object o;
o = c;
```

Променливите c и o са референции на един и същ `Circle` обект (фиг. 3).



Фиг. 3. System.Object референция

да разгледаме още един пример. Ще реализираме абстрактната структура от данни стек:

```
public class Stack
 {
     readonly int size;
     object[] data;
     int sp = 0; // Stack Pointer
     public Stack(int size=10)
     {
         this.size = size;
         data = new object[size];
     }
     public bool Push(object obj)
     {
         if (sp == size) return false;
         data[sp++] = obj;
         return true;
     }
     public bool Pop(out object obj)
     {
         if (sp == 0)
         {
             obj = null;
             return false;
         }
         obj = data[--sp];
         return true;
     }
 }
```

Понеже стекът работи с данни от тип `object`, в него можем да записваме данни от всякакъв тип:

```
Stack stack = new Stack();  // По подразбиране за 10 елемента
stack.Push(new Circle(10)); // обект на Circle
stack.Push(100);            // int
stack.Push("Ivaylo");       // string

object obj;
stack.Pop(out obj);
Console.WriteLine((string) obj);
stack.Pop(out obj);
Console.WriteLine((int) obj);
stack.Pop(out obj);
Console.WriteLine((Circle) obj);
```

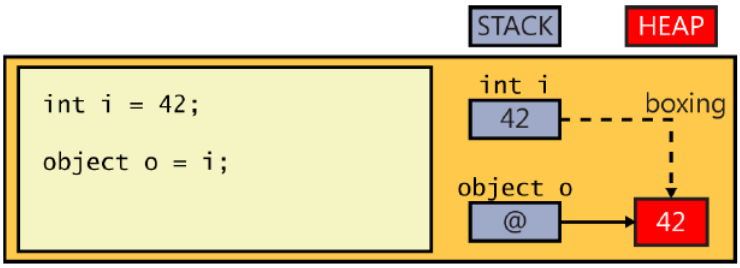
За да могат да се използват, обектите трябва явно да се преобразуват до техния тип. До `object` и обратно от `object` могат да се преобразуват и *value* типове (като `int` в примера). Тази възможност на C# се нарича унифициране на типовете (*type unification*). За да е възможно такова преобразуване виртуалната машина (*Common Language Runtime*) трябва да извърши малко специална работа, за да заобиколи разликите в семантиката между стойностни и референтни типове. Този процес се нарича *boxing* и *unboxing*.

Boxing и unboxing

Казахме, че променливи от тип `object` могат да бъдат референции и на стойностни типове.

```
int i = 42;
object o = i;
```

Променливата `i` е от стойностен тип и е разположена в стека. Ако референцията вътре в `o` сочи директно към `i`, то тя ще сочи към област в стека. Обаче всички референции трябва да сочат към обекти в *heap*-а. Създаването на референции към стека може сериозно да застраши стабилността на CLR и да създаде потенциална пробойна в сигурността, затова не е позволено. Вместо това виртуалната машина заделя ново парче памет в *heap*-а и копира в него стойността на `i`, след което насочва референцията на `o` към това копие. Автоматичното копиране на елемент от стека в *heap*-а се нарича *boxing* (фиг. 4).



Фиг. 4. Boxing

Обратното действие – извличане на стойността, сочена от `object` променливата се нарича *unboxing*. То не може да стане директно:

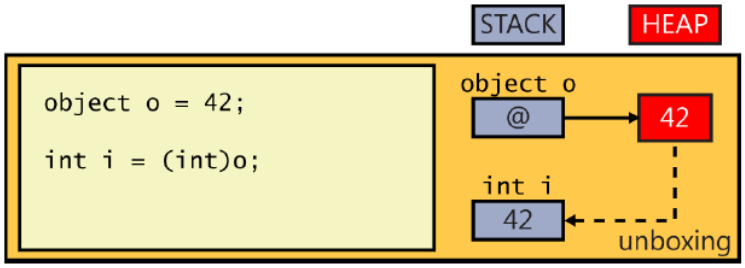
```
int i = o; //error CS0266: Cannot implicitly convert type 'object' to 'int'.
```

Това ограничение има смисъл, защото променлива от тип `object` може да сочи данни от всякакъв тип, който може да не е съвместим с този на променливата отляво на присвояването.

За да извлечем желаната стойност трябва да направим преобразуване на типа (*cast*). Това е операция, която проверява дали е безопасно да се преобразува елемент от един тип в друг преди да се направи копието. Задаваме като префикс желания тип в кръгли скоби

```
int i = 42;
object o = i;
i = (int) o; // OK, unboxing
```

Ефектът от това преобразуване е коварен. Компилаторът забелязва, че сме определили тип `int` за преобразуването, след което генерира код за проверка към какво всъщност сочи `o` в този момент. Това може да бъде абсолютно всичко – само това, че нашият каст казва, че `o` сочи `int`, не означава, че това наистина е така. Ако `o` действително сочи *boxed* `int` и всичко съвпада, преобразуването е успешно и компилаторът генерира код, който извлича стойността от „кутията“ и я копира в `i` (фиг. 5).



Фиг. 5. Unboxing

Ако пък o не сочи *boxed* `int`, имаме разминаване в типовете и преобразуването се проваля, което предизвиква изключение *InvalidCastException* по време на изпълнение на програмата:

```
int i = 42;
object o = new Circle();
try
{
    i = (int) o;
}
catch(InvalidCastException e)
{
    Console.WriteLine(e.Message); // Specified cast is not valid.
}
Console.WriteLine(i);
```

Безопасно преобразуване на типове

Добре е да се прихваща изключението, като в горния пример, но е трудно да се предприемат действия, ако типът на обекта не е този, който очакваме. Това е тромав подход. C# предоставя още две много полезни операции, които могат да помогнат за по-елегантно преобразуване на типовете. Това се операциите `is` и `as`.

```
object o = new Circle(42);
Circle c = null;
if (o is Circle)
    c = (Circle) o;
Console.WriteLine(c?.Area());
```

Операцията `is` има два операнда – референция на обект отляво и име на тип отдясно. Ако обектът, референсиран в *heap*-а е от съответния тип, операцията връща стойност `true`, в противен случай – `false`. Така можем да извършим преобразуването, само ако знаем, че ще е успешно.

Операцията `as` изпълнява подобна роля – тя също има два операнда (обект и тип). Виртуалната машина прави опит да преобразува обекта до желания тип. Ако преобразуването е успешно, върнатата стойност е преобразувания обект. В противен случай връща стойност `null`.

```
object o = new Circle(42);
Circle c = o as Circle;
Console.WriteLine(c?.Area());
```

В този пример ако преобразуването на o до `Circle` е успешно, обектът ще се присвои на `c`. В противен случай c ще се инициализира с `null`.

Като учим наследяване, ще се върнем на още някои особености на операциите `is` и `as`... Като в този пример:

```
object o = new Cilinder(42, 20); // цилиндър с основа окръжност с радиус 42 и ръб 20
Circle c = null;
```

```
if(o is Circle)         // true, защото Circle е базов на Cilinder
    c = o as Circle;
Console.WriteLine(c?.Area());
```

- **Структури**

Видяхме, че класовете дефинират референтни типове, чиито обекти винаги се създават в *heap*-а. В някои случаи класът може да съдържа толкова малко данни, че „разходите“ за поддържането му в *heap*-а са неоправдани. В такива случаи е по-подходящо да се дефинира структура – стойностен тип. Тъй като структурите се съхраняват в стека, докато структурата е разумно малка, управлението на паметта е по-ефективно.

Подобно на класа структурата може да има свои полета, методи, свойства и конструктори, но има наложени някои ограничения, които ще разгледаме по-долу.

Тук е мястото да споменем, че всички примитивни числови типове като `int`, `long` и `float` са псевдоними съответно на структурите `System.Int32` , `System.Int64` и `System.Single`. Тези структури имат полета и методи и ние можем да извикваме методи върху променливи и литерали от тези типове. Например всички тези структури предоставят метода `ToString()`, който преобразува числовата стойност в низ. Следните оператори са напълно коректни:

```
int i = 55;
Console.WriteLine(i.ToString());
Console.WriteLine(55.ToString());
float f = 98.765F;
Console.WriteLine(f.ToString());
Console.WriteLine(98.765F.ToString());
```

Методът `Console.WriteLine()` извиква метода `ToString()`, когато е необходимо, така че горният пример не е много подходящ. Подходящо обаче е използването на статичните методи, които тези структури предоставят, например методът `Parse()` за преобразуване на низ в число.

```
string s = "42";
i = int.Parse(s); // System.Int32.Parse(s);
```

Тези структури съдържат и някои полезни статични полета, например `Int32.MaxValue` и `Int32.MinValue`.

Деклариране на структури

Декларирането на структура е подобно на декларирането на клас.

```
public struct Time
{
    public int hours;
    public int minutes;
    public int seconds;
}
```

Както и при класовете, в повечето случаи не е препоръчително полетата да са `public`. Тук например няма как да контролираме дали данните, записани в тях са коректни. По-добрият вариант е полетата да са `private`, а `public` конструктори и методи да се грижат за тяхното манипулиране.

```
public struct Time
{
    private int hours, minutes, seconds;
    public Time(int hh, int mm, int ss=0)
    {
```

```
            this.hours = Math.Abs(hh) % 24;
            this.minutes = Math.Abs(mm) % 60;
            this.seconds = Math.Abs(ss) % 60;
        }
    }
```

Или вариант без конструктор – с `public` свойства:

```
public struct Time
{
    private int hours, minutes, seconds;
    public int Hours { get => hours; set => hours = Math.Abs(value) % 24; }
    public int Minutes { get => minutes; set => minutes = Math.Abs(value) % 60; }
    public int Seconds { get => seconds; set => seconds = Math.Abs(value) % 60; }
}
```

В този случай при дефинирането на променлива от този тип може да се използва обектен инициализатор:

```
Time t1 = new Time { Hours = 36, Minutes = 15, Seconds = 0 };
```

Без допълнително писане на код не могат да се извършват много от често използваните операции, например не могат да се сравняват две структури с операциите == и !=, но може да се използва достъпният за всички структури метод `Equals()` и, както ще видим по-късно в курса, могат да се предефинират операции, така че да работят и с потребителски дефинираните типове.

Когато се копират променливи от стойностни типове получаваме две копия на стойностите (два различни обекта), а при копиране на променливи от референтни типове получаваме две референции към един и същ обект. Затова структурите са подходящи за малки по обем данни, за които е почти еднакво ефективно копирането на стойността и на адреса (референцията), а класовете – за по-сложни данни, които са твърде обемни, за да се копират ефективно.

Разлики между класове и структури

Ще формулираме накратко разликите:

- структурата е стойностен тип, а класът – референтен;
- структурите не поддържат наследяване (макар, че наследяват `System.ValueType`);
- структурите не могат да имат
  - подразбиращ се конструктор (без параметри);
  - инициализатори на полетата;
  - финализатор;
  - виртуални или `protected` методи.

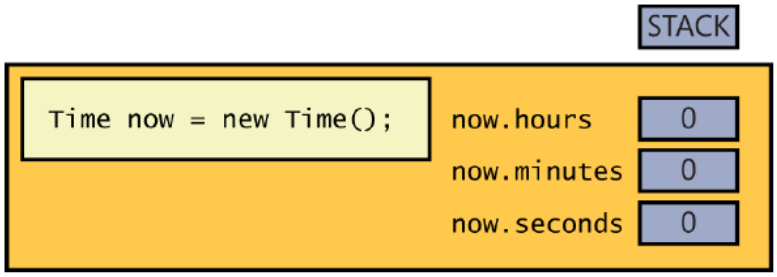
Подразбиращият се конструктор, който създава компилаторът и не може да бъде предефиниран, инициализира всички полета със стойности `0`, `false` или `null` (също както и при класовете). Ако дефинираме конструктор (с параметри) той трябва да инициализира всички полета (които, казахме, не могат да имат и инициализатори).

Инициализиране на структури

Инициализирането на структура може да стане чрез конструктор:

```
Time now = new Time();
```

Подразбиращият се конструктор ще инициализира полетата с нули и ситуацията ще изглежда като на Фиг. 6.



Фиг. 6 Инициализация на структура по подразбиране

Тъй като структурата е стойностен тип можем да създадем променлива от този тип без да извикваме конструктор:

```
Time now;
```

В този случай променливата се създава, но полетата ѝ остават в неинициализирано състояние. Всеки опит за достъп до полетата води до грешка по време на компилация.

За инициализацията можем да използваме и собствен конструктор, като описания по-горе.

```
Time now = new Time(12,30);
```

Този конструктор трябва да инициализира всички полета. В случая обръщението използва подразбиращата се стойност на параметъра `ss`, за да инициализира секундите с `0`.

Копиране на структури

Възможно е на променлива от тип структура да се присвоява друга променлива от същия тип, както и една променлива-структура да се инициализира с друга. За целта променливата отдясно трябва да е напълно инициализирана (всичките ѝ полета да имат зададени стойности).

```
Time t1 = new Time();
Time t2 = t1;   // OK, t1 е инициализирана;
Time t3;
Time t4 = t3;   // Грешка! t3 не е напълно инициализирана
```

Копирането се различава от това при класовете. На всяко поле на структурата отляво се присвоява стойността на съответното поле на структурата отдясно. Това присвояване се изпълнява като една единствена операция по копиране на цялото съдържание на структурата и никога не предизвиква изключение. В резултат двете променливи референсират различни обекти в паметта.

```
Time t1 = new Time(7,30,0);       // Time е структура
Time t2 = t1;                     // OK, t1 е инициализирана;
t1.Hours = 11;                    // променяме t1
Console.WriteLine($"t1: {t1}");   // t1: 11h 30m 00s
Console.WriteLine($"t2: {t2}");   // t2 остава непроменена  (7:30)
```

Ако `Time` беше реализиран като клас, подобно копиране щеше да означава създаването на втора референция към същия обект.

```
Time t1 = new Time(7,30,0);         // Time е клас
Time t2 = t1;                       // OK, t1 е инициализирана;
t1.Hours = 11;                      // променяме t1
Console.WriteLine($"t1: {t1}");     // t1: 11h 30m 00s
Console.WriteLine($"t2: {t2}");     // t2 също е променена
```