Лекция №2. Базови елементи на езика

- **Пространства на имената (*Namespaces*)**

Чрез тях .NET осигурява контейнери за код на приложенията, така че съдържащите се в този код елементи да могат да бъдат еднозначно идентифицирани. Пространствата на имената се използват още и за категоризиране на елементите на .NET Framework, повечето от които са дефиниции на типове (например `System.Int32`, `System.String`).

По подразбиране кодът се разполага в глобалното пространство на имената (*global namespace*) и е достъпен само по име (без квалифициране). С помощта на ключовата дума `namespace` можем да дефинираме *namespace* за блок от код, заграден във фигурални скоби. Имената от това пространство трябва да бъдат квалифицирани, ако се използват извън него. Квалифицираните имена използват точка (`.`) за разделител между различните нива на влагане на пространствата

```
System.Collections.Generic.List<string> list = new List<string>();
```

С помощта на оператора `using` може да се ползват имената от дадено пространство, ако неговият код е свързан по някакъв начин към проекта (или е дефиниран в сорс код към проекта, или е включен в References на проекта)

```
using System.Collections.Generic;
List<string> list;
```

Visual Studio създава *namespace* за всеки нов проект.

От C# 6 включително е наличен операторът `using static`. С него се осигурява директен достъп до статични членове, например статичният метод `public static void WriteLine`, който е част от статичния клас `System.Console`, може да бъде извикван директно с краткото си име:

```
using static System.Console;
WriteLine("It's OK");
```

- **Преобразуване на типовете**

Всички данни, независимо от своя тип, се записват в поредици от битове (нули и единици). Типът казва как да се интерпретират тези битове. Например типът `char` (`System.Char`) използва 16 бита, за да представи *unicode* символ чрез число между 0 и 65535. Това число се записва по същия начин, както и неотрицателно цяло число от типа `ushort` (`System.UInt16`). По принцип обаче различните типове използват различни схеми за представяне на данните, което означава, че дори да е възможно да разположим поредица от битове в променлива от друг тип, не винаги ще получим това, което очакваме. Затова вместо директно съпоставяне на редици от битове трябва да използваме преобразуване на типовете. Има два вида преобразуване – косвено (*implicit*) и явно (*explicit*).

При косвеното преобразуване от тип A към тип B е възможно във всички случаи и правилата за такова преобразуване са достатъчно прости, за да се доверим на компилатора:

```
char ch ='a';
ushort us = ch;          //implicit conversion
Console.WriteLine(us);   //a
us = 1048;
ch = (char) us;          //explicit conversion
Console.WriteLine(ch);   //и
```

Виждаме, че косвено преобразуване от `ushort` към `char` не е възможно.

Implicit Numeric Conversions Table: https://msdn.microsoft.com/en-us/library/y5b434w4.aspx

From	To
sbyte	short, int, long, float, double, or decimal
byte	short, ushort, int, uint, long, ulong, float, double, or decimal
short	int, long, float, double, or decimal
ushort	int, uint, long, ulong, float, double, or decimal
int	long, float, double, or decimal
uint	long, ulong, float, double, or decimal
long	float, double, or decimal
char	ushort, int, uint, long, ulong, float, double, or decimal
float	double
ulong	float, double, or decimal

Правилото за косвено преобразуване е следното: Всеки тип A, чийто диапазон на допустими стойности попада изцяло в диапазона допустими стойности на тип B, може да бъде косвено преобразуван до тип B.

Синтаксисът за явно преобразуване е следният:

```
(<destinationType>) <sourceVar>
```

Преобразува стойността на `<sourceVar>` към тип `<destinationType>`.

При такова преобразуване е възможна загуба на данни:

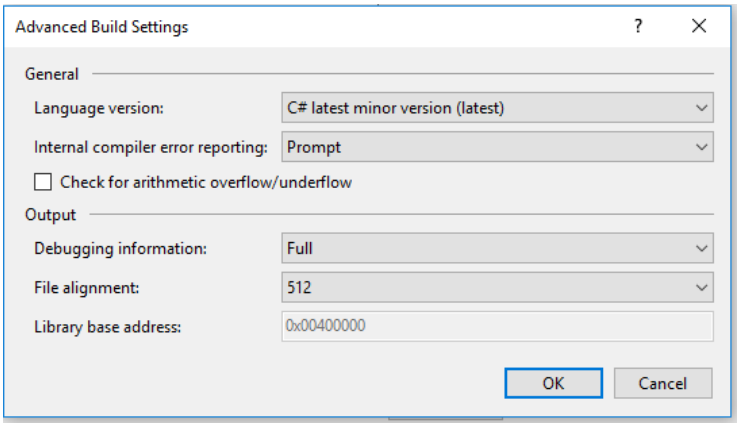
```
byte dest;          //8-bit unsigned integer
short source = 281; //16-bit signed integer
dest = (byte) source;
Console.WriteLine($"source: {source}");      //source: 281
Console.WriteLine($"destination: {dest}");  //destination: 25
```

Програмистът трябва да се погрижи да провери дали стойността на `<sourceVar>` може да се събере в `<destinationType>` или да накара системата да провери за препълване (*overflow*) по време на изпълнение на програмата. Това става с ключовата дума `checked`.

```
dest = checked ((byte) source);
```

Изпълнението на този код ще предизвика грешка „An unhandled exception of type 'System.OverflowException' occurred…„. Ако използваме вместо това ключовата дума `unchecked`, програмата ще се държи както в по-горния пример (без проверка).

Могат да се променят настройките на проекта за проверка за препълване (*overflow checking*), така че по подразбиране всеки такъв израз да е `checked`, освен ако изрично не се укаже `unchecked`. С десния бутон върху името на проекта в Solution Explorer, Properties/Build/, бутона Advanced… и избираме чекбокса Check for arithmetic overflow/underflow:



Фиг. 1 Advanced Build Settings

Явно преобразуване може да се прави и с командите за преобразуване от статичния клас `System.Convert`. Сигурно вече сте ги ползвали при въвеждане на числови данни от клавиатурата:

```
string snum = Console.ReadLine();
double num = System.Convert.ToDouble(snum);
Console.WriteLine(num);
```

Ясно е, че такива преобразувания няма да работят с всякакви въведени стойности. В случая въведеният низ трябва да отговаря на изискванията за записване на числа – може да има знак и да е в експоненциална форма.

Характерна особеност е, че такива преобразувания винаги са `checked` и ключовите думи, както и настройките на проекта не влияят на това.

- **Функции**

Сигнатура на функция – името и параметрите, без типа на резултата.

Досега сте работили само със статични функции.

Имената на функциите обикновено се пишат по конвенцията *PascalCase*.

Кратки функции, които изпълняват например един ред код могат да се пишат като *expression-bodied methods* (възможност, налична в C# 6). Използва се => (*lambda arrow*).

```
public static int Product(int a, int b) => a * b;
```

вместо

```
public static int Product(int a, int b)
{
    return a * b;
}
```

Общият вид на дефиниция на функция е следният:

```
[<static>] <тип_на_резултата> <Име>(<тип> <параметър>, ...)
{
        ...
        return <стойност>;
}
```

Параметри:

В спецификацията на езика C# се разграничават понятията „параметър“ и „аргумент“. Параметрите са част от дефиницията на функцията, а аргументите – част от обръщението към нея (извикването ѝ). Някъде се наричат съответно формални и фактически параметри. Те трябва да си съответстват (по типове, брой и подредба).

Всеки параметър е достъпен в тялото на функцията като променлива.

Пример: Намиране на най-голямата стойност в масив от цели числа.

```
static int MaxValue(int[] intArray)
{
    int max = intArray[0];
    for (int i = 1; i < intArray.Length; i++)
    {
        if (intArray[i] > max)
            max = intArray[i];
    }
    return max;
}

static void Main(string[] args)
{

    int[] myArray = { 1,8,3,6,2,5,9,3,0,2 };
    int maxVal = MaxValue(myArray);
    Console.WriteLine($"The maximum value in myArray is {maxVal}");
    // Може и така:
    Console.WriteLine($"{MaxValue(new int[] { 1,8,3,6,2,5,9,3,0,2 })}");
}
```

Функции с променлив брой аргументи

C# позволява функцията да съдържа точно един специален параметър, който трябва да бъде последен в списъка с параметри. Той се нарича *parameter array* и се разглежда като масив от параметри. Този параметър се отбелязва с ключовата дума `params`. С негова помощ една и съща функция може да бъде извиква с различен брой аргументи.

Ако променим сигнатурата на функцията `MaxValue`, като добавим пред параметъра ѝ `params`,

```
static int MaxValue(params int[] intArray)
```

ще можем да я извикваме така:

```
int maxVal = MaxValue(1, 8, 3, 6, 2, 5, 9, 3, 0, 2);
```

Тази функция може да бъде извиквана и с аргумент масив, както в предишния ѝ вид:

```
int maxVal = MaxValue(myArray);
```

Още един пример – функция, която намира сумата от аргументите си – цели числа:

```
static int SumInts(params int[] intArray)
{
    int sum = 0;
    foreach (int x in intArray)
        sum += x;
    return sum;
}
```

Тя се извиква така:

```
Console.WriteLine(SumInts(1, 2, 3, 4, 5));                  //15
Console.WriteLine(SumInts(1, 8, 3, 6, 2, 5, 9, 3, 0, 2));   //39
Console.WriteLine(SumInts(myArray));                        //39
```

Предаване на параметрите по референция и по стойност

В примерите дотук параметрите на функциите са параметри-стойности (*value parameters*). Това означава, че стойността на аргумента (с който се извиква функцията) се копира в променливата, която използваме за параметър и функцията работи с това копие, а не с оригиналния параметър. Всякакви промени на параметъра не се отразяват на аргумента, например:

```
static void Swap(int x, int y)
{
    int t = x;
    x = y;
    y = t;
}

static void Main()
{
    int a = 5, b = 10;
    Swap(a, b);
    Console.WriteLine($"a = {a}, b = {b}");       // interpolated string expression
    Console.WriteLine("a = {0}, b = {1}", a, b);  // вместо да се използва това
}
```

За да се извърши размяната, параметрите трябва да се предадат по референция, тоест функцията да работи с обектите, с които е била извикана, а не с техни копия. Това става със спецификатора `ref` за съответния параметър и аргумент:

```
static void Swap(ref int x, ref int y)
```

Извикването в `Main()` също изисква спецификатора:

```
Swap(ref a, ref b);
```

Предаването на параметър по референция има две ограничения:

- аргументът не може да бъде константа;
- аргументът трябва да бъде **инициализирана** променлива.

out параметри

Това е още една възможност за управление на начина, по който се предават параметрите по референция. Използва се модификатора `out`. Той е подобен на `ref`, защото отново в края на изпълнението на функцията стойността на параметъра се записва в променливата-аргумент, но има две съществени разлики:

- може да се използва неинициализирана променлива като *out* аргумент;

- функцията третира *out* параметъра като неинициализиран. Тоест дори аргумента да има някаква стойност в момента на извикването, тази стойност се губи, когато започне изпълнението на функцията.

Пример: Ще променим функцията `MaxValue()`, така че да намира и индекса на елемента с най-голяма стойност. Ще променим малко и алгоритъма, като използваме полето `MinValue` на типа `int`.

```
static int MaxValue(int[] intArray, out int maxIndex)
{
     int max = int.MinValue; //най-малката стойност, допустима за типа
     maxIndex = 0;
     for (int i = 0; i < intArray.Length; i++)
     {
         if (intArray[i] > max)
         {
             max = intArray[i];
             maxIndex = i;
         }
     }
     return max;
 }
```

И в `Main`:

```
int[] myArray = { 1, 8, 3, 6, 2, 5, 9, 3, 0, 2 };
int maxIndex;
Console.WriteLine($"The maximum is {MaxValue(myArray, out maxIndex)}");
Console.WriteLine($"The first occurrence of this value is at pos {maxIndex + 1}");
```

Отново при извикването се добавя спецификатора `out`.

Декларациите на `out` променливи могат да се „инлайнват“, тоест променливата да се декларира в списъка с аргументи на функцията (при нейното извикване). Това обединява декларацията и инициализацията в един оператор. Дори и така дефинирана, променливата е „видима“ в целия блок, а не само в оператора, където е използвана. За нашия пример функцията `Main()` ще изглежда така:

```
int[] myArray = { 1, 8, 3, 6, 2, 5, 9, 3, 0, 2 };
Console.WriteLine($"The maximum is {MaxValue(myArray, out int maxIndex)}");
Console.WriteLine($"The first occurrence of this value is at pos {maxIndex + 1}");
```

За да се възползваме от този синтаксис трябва да използваме версия C#7 или по-късна. Добра идея е да настроим в *Build* опциите на проекта (фиг. 4) в полето Language version: “C# latest minor version (latest)”. Така винаги ще използваме най-новата версия на езика.

В практиката често се случва да извикваме функция (обикновено библиотечна), която има множество `out` параметри, а ние не се нуждаем от всичките. За да се изпълни функцията обаче към нея трябва да се подадат аргументи към всичките параметри (тоест за всеки `out` параметър трябва да дефинираме променлива, независимо дали ни трябва съответната стойност). C#7 допуска вместо такава променлива да се използва *discard* – *write-only* променлива, чието име е `_`. На тази единствена променлива можем да присвояваме всички стойности, които желаем да отхвърлим (нямаме нужда от тях). Тази променлива може да се използва само в обръщение към функция или в оператор за присвояване.

Да предположим, че имаме последната версия на функцията `MaxValue()`, но не се интересуваме от индекса, а само от стойността на максималния елемент, която искаме да присвоим на променливата `max`. За `out` аргумент използваме *discard*:

```
var max = MaxValue(myArray, out _);
```

Референции към променливи, дефинирани другаде (*Ref locals and returns*)

Да се върнем на примера: Знаейки позицията, на която се намира търсеният елемент, ние можем да променяме неговата стойност.

```
myArray[maxIndex] = 100;
foreach(var e in myArray)
    Console.Write($"{e} ");
Console.WriteLine();
```

Стойността на най-големия елемент на масива вече е сменена на 100.

Можем да получим референция към този елемент и без да използваме индекса му в клиентския код. Променяме още веднъж функцията `MaxValue()`.

```
static ref int MaxValue(int[] intArray)
{
    int maxIndex = 0;
    for (int i = 1; i < intArray.Length; i++)
        if (intArray[i] > intArray[maxIndex])
            maxIndex = i;
    return ref intArray[maxIndex];
}
```

Резултатът, който тя връща сега е референция на оригиналния елемент (максималния) в масива, а не копие, съдържащо същата стойност. Това се прави с добавяне на ключовата дума `ref` към типа на резултата и към всеки `return` израз.

Променяме и функцията `Main()`:

```
static void Main(string[] args)
{
    int[] myArray = { 1, 8, 3, 6, 2, 5, 9, 3, 0, 2 };
    ref int maximum = ref MaxValue(myArray);
    Console.WriteLine($"The maximum is {MaxValue(myArray)}");
    maximum = 100;
    foreach (var e in myArray)
        Console.Write($"{e} ");
    Console.WriteLine();

}
```

Променливата `maximum` също е декларирана с модификатор `ref`, тоест за нея не се заделя нова памет, а тя ще сочи към място, заето от друга променлива. Това място се получава от обръщението към функцията `MaxValue()`. Обръщението към нея също изисква ключовата дума `ref`. Сега, променяйки стойността на променливата `maximum`, променяме директно стойността, записана в масива, което може да се види при изпълнението на кода. Функцията `MaxValue()` може да се извиква и без модификатора `ref`, ако е необходимо само копие на стойността на максималния елемент, а не референция към него.

```
Console.WriteLine($"The maximum is {MaxValue(myArray)}");
int max = MaxValue(myArray);
```

В този пример променливата `max` не е референция на елемента в масива, а самостоятелна променлива, която няма връзка с него. Всякакви промени на `max` няма да влияят на елементите на масива.

Функции, които връщат референции могат да се използват и като леви стойности – да стоят отляво на операцията за присвояване. В нашия пример можем да променим максималния елемент на масива и без да използваме допълнителна променлива:

```
MaxValue(myArray) = 200;
foreach (var e in myArray)
    Console.Write($"{e} ");
Console.WriteLine();
```

in параметри и ref readonly върнати стойности от функции

В C# 7.2, освен модификаторите out и ref за предаване на параметри по референция, с цел избягване на ненужно копиране (което, както видяхме по-горе, винаги се случва при предаване по стойност), е добавен модификаторът in. За тези, които познават C++, това е аналог на предаването на параметър чрез константен псевдоним. in параметрите се предават по референция, но функцията няма право да променя техните стойности. Така аргументите не се копират и се подобрява ефективността на кода. Ще дадем само илюстративен пример, който в реална ситуация едва ли ще доведе до някакво подобрение:

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

Обръщението към такива функции, за разлика от тези с ref и out аргументи, не изисква използване на in. Също така е допустимо извикването с константи и/или литерали.

```
int x = 5, y = 6;
const int z = 10;
Console.WriteLine(Sum(x,y));    //извикване с променливи
Console.WriteLine(Sum(1,2));    //извикване с числови литерали
Console.WriteLine(Sum(z, 2));   //извикване с константа и числов литерали
```

Пак от съображения за икономия на памет и избягване на ненужно копиране има възможност функция да върне по референция резултат от стойностен тип (*value type*), но да забрани на клиентския код да модифицира тази стойност. Това се обявява в декларацията на функцията с модификатора ref readonly към типа на резултата.

Да се върнем на примера с максималния елемент на масива. За да осигурим защита на елемента в него, само заглавният ред на функцията трябва да се промени:

```
static ref readonly int MaxValue(int[] intArray)
```

В този случай обаче можем да използваме върнатия от функцията резултат само като дясна стойност:

```
ref readonly int maximum = ref MaxValue(myArray);
Console.WriteLine($"The maximum is {MaxValue(myArray)}");
maximum = 100;  // Грешка! Променливата е readonly.
MaxValue(myArray) = 200;   // Грешка!
```

Лесно можем да направим копие на ref readonly върнатата стойност. Трябва просто да я присвоим на променлива, която не е декларирана като ref readonly:

```
int max = MaxValue(myArray);
```

Предефиниране на функции

Това е възможността да имаме няколко функции с едно и също име, но с различни параметри (по брой и тип). Коя функция да бъде изпълнена решава компилаторът в зависимост от аргументите, с които е извикана.

```
static int Sum(int a, int b)
{
    return a + b;
}
```

```
static double Sum(double a, double b)
{
    return a + b;
}
static int Sum(params int[] args)
{
    int s = 0;
    foreach (int x in args)
        s += x;
    return s;
}
static void Main(string[] args)
{
    Console.WriteLine(Sum(1,2));            // 3
    Console.WriteLine(Sum(1,2,3,4,5));      // 15
    Console.WriteLine(Sum(3.5, 4.5));       // 8
}
```

Възможно е функциите да се различават само по това дали параметрите им се предават по стойност или по референция:

```
static void ShowDouble(ref int val)
{
    //...
}
static void ShowDouble(int val)
{
    //...
}
```

Коя от двете версии ще се изпълни зависи дали в обръщението се съдържа `ref`.

Параметри с подразбиращи се стойности (*Optional Parameters*)

Позволява се асоцииране на параметъра с **константна** стойност в декларацията на метода. Това позволява извикване на метода с пропускане на подразбиращите се параметри.

```
public static int Power(int a, int n=2)
{
    int res = a;
    for (int i = 1; i < n; i++)
    {
        res *= a;
    }
    return res;
}
static void Main(string[] args)
{
    Console.WriteLine(Power(2,3));  // 8
    Console.WriteLine(Power(5));    // 25
}
```

Параметрите с подразбиращи се стойности трябва да са разположени след изискваните параметри (тези, които нямат подразбиращи се стойности).

Именувани аргументи

При извикването на функцията може изрично да се посочи името на параметъра, на който да се присвои стойността, а не да се разчита на съответствие в подредбата на аргументите и параметрите. Това е удобно при функции с много параметри с подразбиращи се стойности – така

можем да прескочим няколко от тях. Недостатък е, че при промяна на името на параметъра трябва да се променя и клиентският код. Затова се препоръчва да не се преименуват параметри.

```
static void Main(string[] args)
{
    DisplayGreeting(lastName: "Donchev", firstName:"Ivaylo");

}
public static void DisplayGreeting(
    string firstName,
    string middleName = default(string), // или само default (C# 7.1)
    string lastName = default(string))
{
    Console.WriteLine($"Hello, {firstName} {middleName} {lastName}!");
}
```

- **Делегати**

Делегатът (*delegate*) е тип, чиито стойности са референции към функции. Типичното приложение на делегатите е при работата със събития (*event handling*), която ще разгледаме по-късно в курса. Декларирането на делегат е подобно на това на функция, но без тяло. Използва се ключовата дума `delegate`. Декларацията определя тип на резултата и списък с параметри на делегата.

След декларирането на делегата могат да се декларират променливи от тип този делегат и те да се инициализират с референции на функции, които имат същия тип на резултата и списък с параметри. След това променливата-делегат може да се използва като функция. Това позволява чрез делегати функции да се предават като параметри на други функции.

Пример: Извикване на библиотечни и потребителски функции чрез делегат:

```
class Program
{
   delegate double Calculation(double a);  // декларация на делегат
   static double Square(double x) => x * x;
   static void Proccess(Calculation calc, double arg)  // функция с аргумент делегат
   {
       Console.WriteLine(calc(arg));   // извикване на ф-я чрез променлива-делегат
   }
   static void Main()
   {
       Console.WriteLine("Enter the angle in degrees: ");
       double x = Convert.ToDouble(Console.ReadLine()) * Math.PI / 180;
       Proccess(Math.Sin, x);
       Proccess(Math.Cos, x);
       Console.WriteLine("Enter the number to square: ");
       x = Convert.ToDouble(Console.ReadLine());
       Proccess(Square, x);    // потребителската функция като параметър

       Calculation c;  // дефиниране на променлива-делегат
       c = n => 2 * n; // ламбда функция, която пасва на делегата
       Console.WriteLine(c(x));   // 2*x

       Console.ReadKey();
   }
}
```

Присвояването на референция на функция на променливата-делегат може да стане и така:

```
c = new Calculation(n=>2*n);
c = new Calculation(Math.Sin);
```

Въпрос на личен избор е кой от двата синтаксиса да се използва.

- **Изброявания (*enums*)**

Има ситуации, в които бихме искали да ограничим множеството от стойности, които може да заема дадена променлива, както и да зададем имена на тези стойности. В такива случаи е удачно да се използва изброяване.

Пример: Посоките на света.

```
enum Orientation { North, South, East, West }
```

Изброяването дефинира тип от краен брой стойности, които ние задаваме. След това могат да се дефинират променливи от този тип и да им се присвояват стойности от изброяването.

```
Orientation orientation = Orientation.North;
```

Изброяванията имат базисен тип, използван за съхраняване на стойностите. По подразбиране той е `int`. Може да се избере друг базисен тип при деклариране на изброяването:

```
enum <typeName> : <underlyingType>
{
    <value1>,
    <value2>,
    <value3>,
    ...
    <valueN>
}
```

```
enum Orientation : byte { North, South, East, West }
```

Базисни типове могат да бъдат `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, и `ulong`. По подразбиране на всяка стойност на изброяването се присвоява кореспондираща стойност на базисния тип, започвайки от нула. С оператора = могат да се задават други стойности. Стойностите може да се повтарят. Ако при тези присвоявания се получи цикъл, това предизвиква грешка:

```
enum <typeName> : <underlyingType>
{
     <value1> = <value2>,
     <value2> = <value1>
}
```

Възможно е преобразуване на типовете от изброяване към базисен и обратно:

```
Orientation orientation = (Orientation) 2;
Console.WriteLine(orientation);     // East
orientation = Orientation.South;
Console.WriteLine((byte)orientation); // 1
```

Възможно е и преобразуване от `string` до изброяване. Използва се командата `Enum.Parse`. Синтаксисът е малко по-сложен:

```
Orientation myDirection = (Orientation) Enum.Parse(typeof(Orientation),"West");
```

Трябва да се внимава, защото ако се използва низ, който не съвпада със стойност на изброяването, ще възникне грешка. Тези стойности са *case sensitive*.

- **Switch изрази**

Това е нововъведение в C#8, което позволява по-сбит и интуитивен синтаксис – с по-малко повторения на ключовите думи `case` и `break` и по-малко скоби. Често операторът `switch` произвежда стойност във всеки от `case` блоковете си. Да разгледаме следния пример: Имаме изброяване с цветовете на дъгата:

```
public enum Rainbow { Red, Orange, Yellow, Green, Blue, Indigo,Violet }
```

Приемаме, че в програмата ни има дефиниран тип `RGBColor`, който конструира цвят от 3 компонента – цели числа. Искаме да преобразуваме цвят от тип `Rainbow` в тип `RGBColor`. С класически оператор `switch` можем да напишем следната функция:

```
public static RGBColor FromRainbowClassic(Rainbow colorBand)
{
	switch (colorBand)
	{
		case Rainbow.Red:		return new RGBColor(0xFF, 0x00, 0x00);
		case Rainbow.Orange:	return new RGBColor(0xFF, 0x7F, 0x00);
		case Rainbow.Yellow:	return new RGBColor(0xFF, 0xFF, 0x00);
		case Rainbow.Green:	return new RGBColor(0x00, 0xFF, 0x00);
		case Rainbow.Blue:		return new RGBColor(0x00, 0x00, 0xFF);
		case Rainbow.Indigo:	return new RGBColor(0x4B, 0x00, 0x82);
		case Rainbow.Violet:	return new RGBColor(0x94, 0x00, 0xD3);
		default: throw new ArgumentException(message: "invalid enum value",
				paramName: nameof(colorBand));
	}
}
```

С помощта на `switch` израз същата функция изглежда така:

```
public static RGBColor FromRainbow(Rainbow colorBand) =>
		colorBand switch
		{
		    Rainbow.Red => new RGBColor(0xFF, 0x00, 0x00),
		    Rainbow.Orange => new RGBColor(0xFF, 0x7F, 0x00),
		    Rainbow.Yellow => new RGBColor(0xFF, 0xFF, 0x00),
		    Rainbow.Green => new RGBColor(0x00, 0xFF, 0x00),
		    Rainbow.Blue => new RGBColor(0x00, 0x00, 0xFF),
		    Rainbow.Indigo => new RGBColor(0x4B, 0x00, 0x82),
		    Rainbow.Violet => new RGBColor(0x94, 0x00, 0xD3),
		    _ => throw new ArgumentException(message: "invalid enum value",
		            paramName: nameof(colorBand)),
		};
```

Променливата е преди ключовата дума `switch`, по което лесно различаваме израза от `switch` оператора. Елементите `case` и `:` са заменени от `=>`. `default` случая е заменен с *discard*.

```
RGBColor color = FromRainbow(Rainbow.Indigo);
```

- **Индекси и диапазони**

Индексите и диапазоните осигуряват кратък синтаксис за достъп до отделни елементи или диапазони в последователности. В езика са въведени два нови типа и две операции:

- типът `System.Index` представя индекс в последователност;
- операцията „индекс от края“ (*index from end operator*) `^`, определя индекса като относителен спрямо края на последователността;
- типът `System.Range` представя поддиапазон в последователност;
- операцията диапазон (*range*) `..` чрез своите операнди определя началото и края на диапазона .

Нека имаме дефиниран масив `int[] sequence`. Индекс `0` се отнася до елемента `sequence[0]`. Индекс `^0` съответства на `sequence[sequence.Length]`. За всяко число `n` индексът `^n` е същият като `sequence.Length - n`.

Диапазонът се определя от начало и край. Началото попада в диапазона, но краят – не. Така диапазонът `[0..^0]` представя целия диапазон, подобно на `[0..sequence.Length]`.

Да разгледаме следния пример:

```
var words = new string[]
{
                // index from start    index from end
   "The",       // 0                   ^9
   "quick",     // 1                   ^8
   "brown",     // 2                   ^7
   "fox",       // 3                   ^6
   "jumped",    // 4                   ^5
   "over",      // 5                   ^4
   "the",       // 6                   ^3
   "lazy",      // 7                   ^2
   "dog"        // 8                   ^1
};              // 9 (or words.Length) ^0
```

Можем да извлечем последната дума с индекс ^1.

```
Console.WriteLine($"The last word is {words[^1]}"); // writes "dog"
```

Следващият код създава поддиапазон от думите „quick“, „brown“ и „fox“. Елементът `words[4]` не е в диапазона.

```
var quickBrownFox = words[1..4];
```

Следващата декларация създава поддиапазон с думите “lazy” и “dog”. Той включва `words[^2]` и `words[^1]`. Крайният индекс `^0` не попада в диапазона.

```
var lazyDog = words[^2..^0];
```

Още няколко примера за диапазони:

```
var allWords = words[..];       // contains "The" through "dog".
var firstPhrase = words[..4];   // contains "The" through "fox"
var lastPhrase = words[6..];    // contains "the", "lazy" and "dog"
```

Диапазони могат да се дефинират и като променливи:

```
Range phrase = 1..4;
```

След това променливата може да се използва в `[]`.

```
var text = words[phrase];
```

Още един пример:

```
int[] mas = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
for(int i=1; i<=mas.Length; i++)
    Console.Write($"{ mas[^i]} ");  //  index from end operator ^
Console.WriteLine();
```

Обхожда масива и извежда елементите му в обратен ред.

От изучените до момента типове, освен масивите, индекси и диапазони поддържа и `string`.