Краткие сведения о процедурах и функциях

Процедуры и функции - функциональные модули

Первыми формами модульности, появившимися в языках программирования, были *процедуры* и *функции*. Они позволяли задавать определенную функциональность и многократно выполнять один и тот же параметризованный программный код при различных значениях параметров. Поскольку *функции* в математике использовались издавна, то появление их в языках программирования было совершенно естественным. Уже с первых шагов *процедуры* и *функции* позволяли решать одну из важнейших задач, стоящих перед программистами, – задачу повторного использования программного кода. Встроенные в язык *функции* давали возможность существенно расширить возможности языка программирования. Важным шагом в автоматизации программирования было появление библиотек *процедур* и *функций*, доступных из используемого языка.

Процедуры и функции - методы класса

Долгое время *процедуры* и *функции* играли не только функциональную, но и архитектурную роль. Весьма популярным при построении программных систем был метод функциональной декомпозиции "сверху вниз", и сегодня еще играющий важную роль. Но с появлением ООП архитектурная роль функциональных модулей отошла на второй план. Для ООП-языков, к которым относится и язык C#, в роли архитектурного модуля выступает *класс*. Программная система строится из модулей, роль которых играют *классы*, но каждый из этих модулей имеет содержательную начинку, задавая некоторую абстракцию данных.

Процедуры и *функции* связываются теперь с *классом*, они обеспечивают функциональность данных *класса* и называются методами *класса*. Главную роль в программной системе играют данные, а *функции* лишь служат данным. Напомню здесь, что в C# *процедуры* и *функции* существуют только как методы некоторого *класса*, они не существуют вне *класса*.

В данном контексте понятие *класс* распространяется и на все его частные случаи – структуры, интерфейсы, делегаты.

В языке C# нет специальных ключевых слов – **procedure** и **function**, но присутствуют сами эти понятия. Синтаксис объявления метода позволяет однозначно определить, чем является метод – *процедурой* или *функцией*.

Прежнюю роль библиотек *процедур* и *функций* теперь играют *библиотеки классов*. *Библиотека классов* FCL (*библиотеки FCL* – статический компонент Framework .Net.), доступная в языке C#, существенно расширяет возможности языка. Знание *классов* этой *библиотеки* и методов этих *классов* совершенно необходимо для практического программирования на C# с использованием всей его мощи.(Например, *классы* Convert, Math и Random)

Процедуры и функции. Отличия

Функция отличается от *процедуры* двумя особенностями:

- всегда вычисляет некоторое значение, возвращаемое в качестве результата *функции*;
- вызывается в выражениях.

Процедура C# имеет свои особенности:

- возвращает *формальный результат* void, указывающий на *отсутствие результата*;
- вызов *процедуры* является оператором языка;

- имеет *входные* и *выходные аргументы*, причем *выходных аргументов* – ее результатов – может быть достаточно много.

Хорошо известно, что одновременное существование в языке *процедур* и *функций* в каком-то смысле избыточно. Добавив еще один *выходной аргумент*, любую *функцию* можно записать в виде *процедуры*. Справедливо и обратное. Если допускать *функции с побочным эффектом*, то любую *процедуру* можно записать в виде *функции*. В языке C – дедушке C# – так и сделали, оставив только *функции*. Однако значительно удобнее иметь обе формы реализации метода: и *процедуры*, и *функции*. Обычно метод предпочитают реализовать в виде *функции* тогда, когда он имеет один *выходной аргумент*, рассматриваемый как результат вычисления значения *функции*. Возможность вызова *функций* в выражениях также влияет на выбор в пользу реализации метода в виде *функции*. В других случаях метод реализуют в виде *процедуры*.

Описание методов (процедур и функций). Синтаксис

Если переменные хранят некоторые значения, то методы содержат собой набор операторов, которые выполняют определенные действия. По сути метод - это именованный блок кода, который выполняет некоторые действия.

Общее определение методов выглядит следующим образом:

```
[модификаторы] тип_возвращаемого_значения название_метода
([параметры])
{
    // тело метода
}
```

Модификаторы и параметры необязательны.

Например, по умолчанию консольная программа на языке C# должна содержать как минимум один метод - метод Main, который является точкой входа в приложение:

```
static void Main(string[] args)
{

}
```

Ключевое слово static является модификатором. Далее идет тип возвращаемого значения. В данном случае ключевое слово void указывает на то, что метод ничего не возвращает.

Далее идет название метода - Main и в скобках параметры - string[] args. И в фигурные скобки заключено тело метода - все действия, которые он выполняет. В данном случае метод Main пуст, он не содержит никаких операторов и по сути ничего не выполняет.

Определим еще пару методов:

```
using System;

namespace HelloApp
{
    class Program
    {
        static void Main(string[] args)
        {

        }

        static void SayHello()
        {
            Console.WriteLine("Hello");
```

```
15          }
16          static void SayGoodbye()
17          {
18              Console.WriteLine("GoodBye");
19          }
20      }
21  }
```

В данном случае определены еще два метода: SayHello и SayGoodbye. Оба метода определены в рамках класса Program, они имеют модификатор static, а в качестве возвращаемого типа для них определен тип void. То есть данные методы ничего не возвращают, просто производят некоторые действия. И также оба метода не имеют никаких параметров, поэтому после названия метода указаны пустые скобки.

Оба метода выводят на консоль некоторую строку. Причем для вывода на консоль методы используют другой метод, который определен в .NET по умолчанию - Console.WriteLine().

Но если мы запустим даную программу, то мы не увидим никаких сообщений, которые должны выводить методы SayHello и SayGoodbye. Потому что стартовой точкой является метод Main. При запуске программы выполняется только метод Main и все операторы, которые составляют тело этого метода. Все остальные методы не выполняются.

Вызов методов

Чтобы использовать методы SayHello и SayGoodbye в программе, нам надо вызвать их в методе Main.

Для вызова метода указывается его имя, после которого в скобках идут значения для его параметров (если метод принимает параметры).

```
1  название_метода (значения_для_параметров_метода);
```

Например, вызовем методы SayHello и SayGoodbye:

```
1   using System;
2
3   namespace HelloApp
4   {
5       class Program
6       {
7           static void Main(string[] args)
8           {
9               SayHello();
10              SayGoodbye();
11
12              Console.ReadKey();
13          }
14
15          static void SayHello()
16          {
17              Console.WriteLine("Hello");
18          }
19          static void SayGoodbye()
20          {
21              Console.WriteLine("GoodBye");
22          }
23      }
24  }
```

Консольный вывод программы:

```
Hello
GoodBye
```

Преимуществом методов является то, что их можно повторно и многократно вызывать в различных частях программы. Например, в примере выше в двух методах для вывода строки на консоль используется метод Console.WriteLine.

Возвращение значения

Метод может возвращать значение, какой-либо результат. В примере выше были определены два метода, которые имели тип **void**. Методы с таким типом не возвращают никакого значения. Они просто выполняют некоторые действия.

Если метод имеет любой другой тип, отличный от void, то такой метод обязан вернуть значение этого типа. Для этого применяется оператор **return**, после которого идет возвращаемое значение:

```
return возвращаемое значение;
```

Например, определим еще пару методов:

```
static string GetHello()
{
    return "Hello";
}
static int GetSum()
{
    int x = 2;
    int y = 3;
    int z = x + y;
    return z;
}
```

Метод GetHello имеет тип string, следовательно, он должен возвратить строку. Поэтому в теле метода используется оператор return, после которого указана возвращаемая строка.

Метод GetSum имеет тип int, следовательно, он должен возвратить значение типа int - целое число. Поэтому в теле метода используется оператор return, после которого указано возвращаемое число (в данном случае результат суммы переменных x и y).

После оператора return также можно указывать сложные выражения, которые возвращают определенный результат. Например:

```
static int GetSum()
{
    int x = 2;
    int y = 3;
    return x + y;
}
```

При этом методы, которые в качестве возвращаемого типа имеют любой тип, отличный от void, обязательно должны использовать оператор return для возвращения значения. Например, следующее определение метода некорректно:

```
static string GetHello()
{
    Console.WriteLine("Hello");
}
```

Также между возвращаемым типом метода и возвращаемым значением после оператора return должно быть соответствие. Например, в следующем случае возвращаемый тип - int, но метод возвращает строку (тип string), поэтому такое определение метода некорректно:

```
1   static int GetSum()
2   {
3       int x = 2;
4       int y = 3;
5       return "5"; // ошибка - надо возвращать число
6   }
```

Результат методов, который возвращают значение, мы можем присвоить переменным или использовать иным образом в программе:

```
1    using System;
2
3    namespace HelloApp
4    {
5        class Program
6        {
7            static void Main(string[] args)
8            {
9                string message = GetHello();
10               int sum = GetSum();
11
12               Console.WriteLine(message);  // Hello
13               Console.WriteLine(sum);     // 5
14
15               Console.ReadKey();
16           }
17
18           static string GetHello()
19           {
20               return "Hello";
21           }
22           static int GetSum()
23           {
24               int x = 2;
25               int y = 3;
26               return x + y;
27           }
28       }
29   }
```

Метод GetHello возвращает значение типа string. Поэтому мы можем присвоить это значение какой-нибудь переменной типа string: string message = GetHello();

Второй метод - GetSum - возвращает значение типа int, поэтому его можно присвоить переменной, которая принимает значение этого типа: int sum = GetSum();.

Выход из метода

Оператор return не только возвращает значение, но и производит выход из метода. Поэтому он должен определяться после отальных инструкций. Например:

```
1   static string GetHello()
2   {
```

```
3       return "Hello";
4       Console.WriteLine("After return");
5   }
```

С точки зрения синтаксиса данный метод корректен, однако его инструкция Console.WriteLine("After return") не имеет смысла - она никогда не выполнится, так как до ее выполнения оператор return возвратит значение и произведет выход из метода.

Однако мы можем использовать оператор return и в методах с типам void. В этом случае после оператора return не ставится никакого возвращаемого значения (ведь метод ничего не возвращает). Типичная ситуация - в зависимости от определённых условий произвести выход из метода:

```
1    static void SayHello()
2    {
3        int hour = 23;
4        if(hour > 22)
5        {
6            return;
7        }
8        else
9        {
10           Console.WriteLine("Hello");
11       }
12   }
```

Сокращенная запись методов

Если метод в качестве тела определяет только одну инструкцию, то мы можем сократить определение метода. Например, допустим у нас есть метод:

```
1   static void SayHello()
2   {
3       Console.WriteLine("Hello");
4   }
```

Мы можемего сократить следующим образом:

```
1   static void SayHello() => Console.WriteLine("Hello");
```

То есть после списка параметров ставится знак равно и больше чем, после которого идет выполняемая инструкция.

Если у нас есть метод, который возвращает значение:

```
1   static string GetHello()
2   {
3       return "hello";
4   }
```

То возвращаемое значение оборачивается в круглые скобки:

```
1    static string GetHello() => ("hello");
```

Параметры методов

Параметры позволяют передать в метод некоторые входные данные. Например, определим метод, который складывает два числа:

```
1   static int Sum(int x, int y)
2   {
3       return x + y;
```

```
4   }
```

Метод Sum имеет два параметра: x и y. Оба параметра представляют тип int. Поэтому при вызове данного метода нам обязательно надо передать на место этих параметров два числа.

```
1   class Program
2   {
3       static void Main(string[] args)
4       {
5           int result = Sum(10, 15);
6           Console.WriteLine(result);  // 25
7
8           Console.ReadKey();
9       }
10      static int Sum(int x, int y)
11      {
12          return x + y;
13      }
14  }
```

При вызове метода Sum значения передаются параметрам по позиции. Например, в вызове Sum(10, 15) число 10 передается параметру x, а число 15 - параметру y. Значения, которые передаются параметрам, еще называются **аргументами**. То есть передаваемые числа 10 и 15 в данном случае являются аргументами.

Иногда можно встретить такие определения как **формальные параметры** и **фактические параметры**. Формальные параметры - это собственно параметры метода (в данном случае x и y), а фактические параметры - значения, которые передаются формальным параметрам. То есть фактические параметры - это и есть аргументы метода.

Передаваемые параметру значения могут представлять значения переменных или результат работы сложных выражений, которые возвращают некоторое значение:

```
1   class Program
2   {
3       static void Main(string[] args)
4       {
5           int a = 25;
6           int b = 35;
7           int result = Sum(a, b);
8           Console.WriteLine(result);  // 60
9
10          result = Sum(b, 45);
11          Console.WriteLine(result);  // 80
12
13          result = Sum(a + b + 12, 18);//a + b + 12 представляет значение параметра x
14          Console.WriteLine(result);  // 90
15
16          Console.ReadKey();
17      }
18      static int Sum(int x, int y)
19      {
20          return x + y;
```

```
21      }
22  }
```

Если параметрами метода передаются значения переменных, которые представляют базовые примитивные типы (за исключением типа object), то таким переменным должно быть присвоено значение. Например, следующая программа не скомпилируется:

```
1   class Program
2   {
3       static void Main(string[] args)
4       {
5           int a;
6           int b = 9;
7           Sum(a, b);  // Ошибка - переменной a не присвоено значение
8
9           Console.ReadKey();
10      }
11      static int Sum(int x, int y)
12      {
13          return x + y;
14      }
15  }
```

При передаче значений параметрам важно учитывать тип параметров: между аргументами и параметрами должно быть соответствие по типу. Например:

```
1   class Program
2   {
3       static void Main(string[] args)
4       {
5           Display("Tom", 24); // Name: Tom  Age: 24
6
7           Console.ReadKey();
8       }
9       static void Display(string name, int age)
10      {
11          Console.WriteLine($"Name: {name}  Age: {age}");
12      }
13  }
```

В данном случае первый параметр метода Display представляет тип string, поэтому мы должны передать этому параметру значение типа string, то есть строку. Второй параметр представляет тип int, поэтому должны передать ему целое число, которое соответствует типу int.

Другие данные параметрам мы передать не можем. Например, следующий вызов метода Display будет ошибочным:

```
1   Display(45, "Bob"); // Ошибка! несоответствие значений типам параметров
```

Необязательные параметры

По умолчанию при вызове метода необходимо предоставить значения для всех его параметров. Но C# также позволяет использовать необязательные параметры. Для таких параметров нам необходимо объявить значение по умолчанию. Также следует учитывать, что после необязательных параметров все последующие параметры также должны быть необязательными:

```
1   static int OptionalParam(int x, int y, int z=5, int s=4)
```

```
2   {
3       return x + y + z + s;
4   }
```

Так как последние два параметра объявлены как необязательные, то мы можем один из них или оба опустить:

```
1   static void Main(string[] args)
2   {
3       OptionalParam(2, 3);
4
5       OptionalParam(2,3,10);
6
7       Console.ReadKey();
8   }
```

Именованные параметры

В предыдущих примерах при вызове методов значения для параметров передавались в порядке объявления этих параметров в методе. Но мы можем нарушить подобный порядок, используя именованные параметры:

```
1    static int OptionalParam(int x, int y, int z=5, int s=4)
2    {
3        return x + y + z + s;
4    }
5    static void Main(string[] args)
6    {
7        OptionalParam(x:2, y:3);
8
9        //Необязательный параметр z использует значение по умолчанию
10       OptionalParam(y:2, x:3, s:10);
11
12       Console.ReadKey();
13   }
```

Передача параметров в метод по ссылке. Операторы ref и out

В C# значения переменных по-умолчанию передаются по значению (в метод передается локальная копия параметра, который используется при вызове). Это означает, что мы не можем внутри метода изменить параметр из вне:

```
public static void ChangeValue(object a)
{
   a = 2;
}

static void Main(string[] args)
{
   int a = 1;
   ChangeValue(a);
   Console.WriteLine(a); // 1
   Console.ReadLine();
}
```

Чтобы передавать параметры по ссылке, и иметь возможность влиять на внешнюю переменную, используются ключевые слова ref и out.

Ключевое слово ref

Чтобы использовать ref, это ключевое слово стоит указать перед типом параметра в методе, и перед параметром при вызове метода:

```
public static void ChangeValue(ref int a)
{
   a = 2;
}
static void Main(string[] args)
{
   int a = 1;
   ChangeValue(ref a);
   Console.WriteLine(a); // 2
   Console.ReadLine();
}
```

В этом примере мы изменили значение внешней переменной внутри метода.

Особенностью ref является то, что переменная, которую мы передаем в метод, обязательно должна быть проинициализирована значением, иначе компилятор выдаст ошибку «Use of unassigned local variable 'a'». Это является главным отличием ref от out.

Ключевое слово out

Out используется точно таким же образом как и ref, за исключением того, что параметр не обязан быть проинициализирован перед передачей, но при этом в методе переданному параметру обязательно должно быть присвоено новое значение:

```
public static void ChangeValue(out int a)
{
   a = 2;
}
static void Main(string[] args)
{
   int a;
   ChangeValue(out a);
   Console.WriteLine(a); // 2
   Console.ReadLine();
}
```

Если не присвоить новое значение параметру out, мы получим ошибку «The out parameter 'a' must be assigned to before control leaves the current method»

Массив параметров. Ключевое слово params (тема «Массивы»)

Во всех предыдущих примерах мы использовали постоянное число параметров. Но, используя ключевое слово **params**, мы можем передавать неопределенное количество параметров:

```
1   static void Addition(params int[] integers)
2   {
3       int result = 0;
4       for (int i = 0; i < integers.Length; i++)
5       {
```

```
            result += integers[i];
        }
        Console.WriteLine(result);
    }

    static void Main(string[] args)
    {
        Addition(1, 2, 3, 4, 5);

        int[] array = new int[] { 1, 2, 3, 4 };
        Addition(array);

        Addition();
        Console.ReadLine();
    }
```

Причем, как видно из примера, на место параметра с модификатором params мы можем передать как отдельные значения, так и массив значений, либо вообще не передавать параметры. Если же нам надо передать какие- то другие параметры, то они должны указываться до параметра с ключевм словом params:

```
//Так работает
static void Addition( int x, string mes, params int[] integers,)
{}
```

Вызов подобного метода:

```
Addition(2, "hello", 1, 3, 4);
```

Однако после параметра с модификатором params мы НЕ можем указывать другие параметры. То есть следующее определение метода недопустимо:

```
//Так НЕ работает
static void Addition(params int[] integers, int x, string mes)
{}
```

Массив в качестве параметра

Также этот способ передачи параметров надо отличать от передачи массива в качестве параметра:

```
// передача параметра с params
static void Addition(params int[] integers)
{
    int result = 0;
    for (int i = 0; i < integers.Length; i++)
    {
        result += integers[i];
    }
    Console.WriteLine(result);
}
// передача массива
static void AdditionMas(int[] integers, int k)
{
```

```
14      int result = 0;
15      for (int i = 0; i < integers.Length; i++)
16      {
17          result += (integers[i]*k);
18      }
19      Console.WriteLine(result);
20  }
21
22  static void Main(string[] args)
23  {
24      Addition(1, 2, 3, 4, 5);
25
26      int[] array = new int[] { 1, 2, 3, 4 };
27      AdditionMas(array, 2);
28
29      Console.ReadLine();
30  }
```

Так как метод AdditionMas принимает в качестве параметра массив без ключевого слова params, то при его вызове нам обязательно надо передать в качестве параметра массив.