Лекция 2

Программирование C#

Структура программного кода.

Методы. Функции.

Рекурсивные вычисления.

Общая структура программы

```
using System;

namespace HelloApp
{
    class Person
    {

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

        }
    }
}
```

Структура программного кода

- Директива using System разрешает использовать имена стандартных классов из пространства имен ***System*** без указания имени пространства.
- Ключевое слово ***namespace*** создает для проекта **собственное пространство имен**, названное по умолчанию **ConsoleApplication1**.
- Это сделано для того, чтобы можно было давать программным объектам имена, не заботясь о том, что они могут совпасть с именами в других пространствах имен.

- C# — **объектно-ориентированный язык**, поэтому написанная на нем программа представляет собой **совокупность** взаимодействующих между собой **классов**.

- Функция **Main** является главной функцией приложения и точкой входа программы – точнее она является методом класса **Programm**.

- Любая функция в Си-шарп может быть объявлена только в рамках класса, так как C# - полностью объектно-ориентированный язык программирования (ООП).

- Объявление пользовательской функции внутри другой функции (например внутри Main) недопустимо.

```
static void Main(string[] args)
{
    Console.WriteLine("привет мир!");
}
```

Ключевое слово `static` является модификатором.
Статический метод – это метод, который не имеет доступа к полям объекта, и для вызова такого метода не нужно создавать экземпляр (объект) класса, в котором он объявлен.

Далее тип возвращаемого значения - `void` указывает на то, что метод ничего не возвращает. Такой метод еще называется **процедурой**.

Далее идет название метода - Main и в скобках параметры - `string[] args`.
И ниже в фигурные скобки заключено тело метода - все действия, которые он выполняет.

Объявление функции имеет следующую структуру:

[модификатор доступа] [тип возвращаемого значения] [имя функции] ([аргументы])
{
// тело функции
}

```
static  int max( int f, int s )
{
        if ( f > s )
            return f;
        else
            return s;
}
```

- **Функция** являет собой небольшую подпрограмму. Если просто программа - это решение какой-то прикладной задачи, то функция – это тоже решение, только уже в рамках программы и, соответственно, она выполняет задачу «попроще».
- Функции позволяют уменьшить размер программы за счет того, что не нужно повторно писать какой-то фрагмент кода - мы просто вызываем сколько угодно и где нужно объявленную функцию.
- Функции в Си-шарп также называют **методами.**
- От прошлого нам остался еще один термин – «**процедура**». Если у функции тип возвращаемого значения - void - это указывает на то, что функция ничего не возвращает.

- **Функции**
- В отличие от процедур функции возвращают определенное значение. Например, определим пару функций:

```
int Factorial()
{
    return 1;
}

string Hello()
{
    return "Hell to World";
}
```

В функции в качестве типа возвращаемого значения вместо **void** используется любой другой тип. В данном случае тип **int.**

Функции также отличаются тем, что мы обязательно должны использовать оператор **return**, после которого ставится возвращаемое значение.

Возвращаемое значение всегда должно иметь тот же тип, что значится в определении функции.

Так как у нас функция возвращает значение типа **int**, после оператора **return** стоит число **1** - которое неявно является объектом типа **int.**

Оператор return

•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.

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

Между возвращаемым типом метода и возвращаемым значением после оператора **return** должно быть соответствие

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

Оператор return

Оператор **return** не только возвращает значение, но и производит выход из метода.

```
static string GetHello()
{
    return "Hello";
    Console.WriteLine("After return");//эта строка никогда не выполнится
}
```

```
static void SayHello()
{
    int hour = 23;
    if(hour > 22)
    {
        return;
    }
    else
    {
        Console.WriteLine("Hello");
    }
}
```

Можем использовать оператор **return** и в методах с типам **void**, для того чтобы произвести выход из метода в зависимости от определенных условий

```
static void Main(string[] args)
{
    string message = Hello(); // вызов первого
метода

    Console.WriteLine(message);

    Sum(); // вызов второго метода

    Console.ReadLine();
}
static string Hello()
{
    return "Hello to World!";
}
static void Sum()
{
    int x = 2;
    int y = 3;
    Console.WriteLine("{0} + {1} = {2}", x, y, x+y);
}
```

Вызовы функций

Вызовы функций

В примере определены два метода.
Первый метод Hello возвращает значение типа string. Поэтому мы можем присвоить это значение какой-нибудь переменной типа string:

```
string message = Hello(); // вызов первого метода
```

Второй метод - процедура Sum - просто складывает два числа и выводит результат на консоль. Поэтому при вызове мы не можем ее приравнять какой-либо переменной!

```
Sum(); // вызов второго метода
```

Вложенные вызовы функций. Функция как аргумент

```
class Program
{
  public static int GetMax(int[] array)
  {
    int max = array[0];
    for (int i = 1; i < array.Length; i++)
    {
      if (array[i] > max)
        max = array[i];
    }
    return max;
  }
  static void Main(string[] args)
  {
    int[] numbers = { 3, 32, 16, 27, 55, 43, 2, 34 };
    int max;
    max = GetMax(numbers);          //вызов такой функции можно использовать
                                    при присваивании значения

    Console.WriteLine(GetMax(numbers)); // вызов функции также можно использовать
                                        как аргумент при вызове другой
                                        функции. WriteLine() – тоже функция!
    Console.ReadKey();
  }
}
```

Называть функции стоит так, чтобы имя отображало суть функции. Используйте глаголы или словосочетания с глаголами.

Примеры: GetAge(), Sort(), SetVisibility().

Первая строка функции, где указываются тип, имя, аргументы и т.д. называется **заголовком функции**.

```
public static void ReplaceName(string[] names, string name, string newName)
```

[модификатор доступа] [тип возвращаемого значения] [имя функции] ([аргументы])

Аргументы (Параметры) – это те данные, которые необходимы для выполнения функции.

Аргументы записываются в формате : **[тип] [идентификатор].**

Если аргументов несколько, они отделяются запятой.

Аргументы могут отсутствовать.

Параметры (аргументы) методов

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.

```
static void Main(string[] args)
{
    int result = Sum(10, 15);
    Console.WriteLine(result);  // 25

    Console.ReadKey();
}
static int Sum(int x, int y)
{
    return x + y;
}
```

```
static void Main(string[] args)
{
    int a = 25;   int b = 35;
    int result = Sum (a, b);
    Console.WriteLine(result);  // 60

    result = Sum (b, 45);
    Console.WriteLine(result);  // 80

    result = Sum (a + b + 12, 18);
   // "a + b + 12" представляет значение
параметра x
    Console.WriteLine(result);  // 90

    Console.ReadKey();
}
static int Sum (int x, int y)
{
    return x + y;
}
```

Формальные и фактические параметры

Формальные параметры - это собственно параметры метода (в данном случае x и y – то что стоит в заголовке метода),

а фактические параметры - значения, которые передаются формальным параметрам (то, что стоит в вызовах метода).

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

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 перечисления необязательных параметров все последующие параметры в заголовке функции также должны быть необязательными

```
static int OptionalParam(int x, int y, int z=5, int s=4)
{
    return x + y + z + s;
}

static void Main(string[] args)
{
    OptionalParam(2, 3);

    OptionalParam(2, 3, 10);

    Console.ReadKey();
}
```

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

Перегрузка методов - method overloading

- Это возможность создать один и тот же метод, но с разным набором параметров, и в зависимости от имеющихся параметров применять определенную версию метода.
- Создаем в классе несколько методов с одним и тем же именем, но разной сигнатурой. **Сигнатура (англ) или Заголовок** метода складывается из следующих аспектов:
  - Имя метода
  - Количество параметров
  - Типы параметров
  - Порядок параметров
  - Модификаторы параметров

Перегрузка методов . Пример

```
class Calculator
{
    public void Add(int a, int b)
    {
        int result = a + b;
        Console.WriteLine($"Result is {result}");
    }
    public void Add(int a, int b, int c)
    {
        int result = a + b + c;
        Console.WriteLine($"Result is {result}");
    }
    public int Add(int a, int b, int c, int d)
    {
        int result = a + b + c + d;
        Console.WriteLine($"Result is {result}");
        return result;
    }
    public void Add(double a, double b)
    {
        double result = a + b;
        Console.WriteLine($"Result is {result}");
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Calculator calc = new Calculator();
        calc.Add(1, 2); // 3
        calc.Add(1, 2, 3); // 6
        calc.Add(1, 2, 3, 4); // 10
        calc.Add(1.4, 2.5); // 3.9

        Console.ReadKey();
    }
}
```

Вывод на Консоль:

Result is 3 Result is 6 Result is 10 Result is 3.9

Вопрос: в каких случаях пользуются необязательными параметрами, а в каких перегрузкой методов?

Передача параметров по значению

Существует два способа передачи параметров в метод в языке C#: **по значению** и **по ссылке**.

Обычный способ передачи параметров – по значению!

```
static void Main(string[] args)
  {
     int a = 5;
     Console.WriteLine($"Начальное значение переменной a = {a}");

     //Передача переменных по значению
     //После выполнения этого кода по-прежнему a = 5, так как мы передали лишь ее
копию
     IncrementVal(a);
     Console.WriteLine($"Переменная a после передачи по значению равна = {a}");
     Console.ReadKey();
  }
  // передача по значению
  static void IncrementVal(int x)
  {
     x++;
     Console.WriteLine($"IncrementVal: {x}");
  }
```

Передача параметров по ссылке и модификатор ref

```
static void Main(string[] args)
   {
      int a = 5;
      Console.WriteLine($"Начальное значение переменной a  = {a}");
      //Передача переменных по ссылке
      //После выполнения этого кода a = 6, так как мы передали саму
переменную
      IncrementRef(ref a);
      Console.WriteLine($"Переменная a после передачи ссылке равна = {a}");

      Console.ReadKey();
   }
   // передача по ссылке
   static void IncrementRef(ref int x)
   {
      x++;
      Console.WriteLine($"IncrementRef: {x}");
   }
```

В метод **IncrementRef** передается ссылка на саму переменную a в памяти.

И если значение параметра в **IncrementRef** изменяется, то это приводит и к изменению переменной **a**, так как и параметр и переменная указывают на один и тот же адрес в памяти.

Рекурсивные функции

- Рекурсивная функция представляет такую конструкцию, при которой функция вызывает саму себя.
- Возьмем, к примеру, функцию, вычисляющую факториал числа:

```
static int Factorial(int x)
{
    if (x == 0)
    {
        return 1;
    }
    else
    {
        return x * Factorial(x - 1);
    }
}
```

Здесь задается условие, что если вводимое число не равно 0, то мы умножаем данное число на результат этой же функции, в которую в качестве параметра передается число x-1. То есть происходит рекурсивный спуск. И так, пока не дойдем того момента, когда значение параметра не будет равно единице.

При создании рекурсивной функции в ней обязательно должен быть некоторый базовый вариант, который использует оператор return и помещается в начале функции. В случае с факториалом это if (x == 0) return 1;.

И, кроме того, все рекурсивные вызовы должны обращаться к подфункциям, которые в конце концов сходятся к базовому варианту. Так, при передаче в функцию положительного числа при дальнейших рекурсивных вызовах подфункций в них будет передаваться каждый раз число, меньшее на единицу. И в конце концов мы дойдем до ситуации, когда число будет равно 0, и будет использован базовый вариант.

```
static int Fibonachi(int n)
{
    if (n == 0)
    {
        return 0;
    }
    if (n == 1)
    {
        return 1;
    }
    else
    {
        return Fibonachi(n - 1) + Fibonachi(n - 2);
    }
}
```

Другим распространенным показательным **примером рекурсивной функции** служит функция, вычисляющая числа Фиббоначчи.

n-й член последовательности Фибоначчи определяется по формуле: f(n)=f(n-1) + f(n-2), причем f(0)=0, а f(1)=1.

Рекурсия

- Задача: написать программу с рекурсивным методом, которая выводит на экран цифры натурального числа в обратном порядке.