Функции (c#)

- 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

- 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 объектно-ориентированного программирования (ООП) архитектурная роль функциональных модулей отошла на второй план. Для объектно-ориентированных 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 имеют содержательную начинку, задавая некоторую абстракцию данных.

- Прежнюю роль библиотек процедур и функций теперь играют библиотеки классов. Библиотека классов FCL Framework Class Library, библиотека классов каркаса), 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.

- Процедуры и функции связываются с классом, они обеспечивают функциональность данных класса и называются методами класса. Главную роль в программной системе играют данные, а функции лишь служат данным. Напомним еще раз, 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.

Функция отличается от процедуры двумя особенностями:

- 1. Она всегда вычисляет некоторое значение, возвращаемое в качестве результата функции;
- 2. И вызывается в выражениях.

Процедура C# имеет свои особенности:

- 1. Она возвращает формальный результат **void,** указывающий на отсутствие результата;
- 2. Вызов процедуры является оператором языка;
- 3. И она имеет входные и выходные аргументы, причем выходных аргументов - ее результатов - может быть достаточно много.

- 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 процедур и функций (методов класса)

- **заголовок_метода**
  тело_метода

- Рассмотрим синтаксис заголовка метода:
  [атрибуты][модификаторы]{void|
  тип_результата_функции}
- **имя_метода**
  ([список_формальных_аргументов])

- Имя метода и список формальных аргументов составляют сигнатуру метода. Отметим, что в сигнатуру не входят имена формальных аргументов - здесь важны типы аргументов. В сигнатуру не входит и тип возвращаемого результата.

- Модификатор доступа может иметь четыре возможных значения, из которых пока рассмотрим только два - **public** и **private.** Модификатор **public** показывает, что метод открыт и доступен для вызова клиентами и потомками класса. Модификатор **private** говорит, что метод предназначен для внутреннего использования в классе и доступен для вызова только в теле методов самого класса. Отметим, что если модификатор доступа опущен, то по умолчанию предполагается, что он имеет значение **private** и метод является закрытым для клиентов и потомков класса.

- Обязательным при описании заголовка является указание типа результата, имени метода и круглых скобок, наличие которых необходимо и в том случае, если сам список формальных аргументов отсутствует. Формально тип результата метода указывается всегда, но значение **void** однозначно определяет, что метод реализуется процедурой. Тип результата, отличный от **void,** указывает на функцию.

- Вот несколько простейших примеров описания методов:
- **void A() {...};**
  int B(){...};
  public void C(){...};
- Методы **A** и **B** являются закрытыми, а метод **C** - открыт. Методы **A** и **C** реализованы процедурами, а метод B - функцией, возвращающей целое значение.

Список формальных аргументов

- Как уже отмечалось, список формальных аргументов метода может быть пустым, и это довольно типичная ситуация для методов класса. Список может содержать фиксированное число аргументов, разделяемых символом запятой.

- Рассмотрим теперь синтаксис объявления формального аргумента:
- **[ref | out | params] тип_аргумента имя_аргумента.**
- Обязательным является указание типа и имени аргумента. Отметим, никаких ограничений на тип аргумента не накладывается. Он может быть любым скалярным типом

- Несмотря на фиксированное число формальных аргументов, есть возможность при вызове метода передавать ему произвольное число фактических аргументов. Для реализации этой возможности в списке формальных аргументов необходимо задать ключевое слово **params.** Оно задается один раз и указывается только для последнего аргумента списка, объявляемого как массив произвольного типа. При вызове метода этому формальному аргументу соответствует произвольное число фактических аргументов.

- Содержательно, все аргументы метода разделяются на три группы: входные, выходные и обновляемые. Аргументы первой группы передают информацию методу, их значения в теле метода только читаются. Аргументы второй группы представляют собой результаты метода, они получают значения в ходе работы метода. Аргументы третьей группы выполняют обе функции. Их значения используются в ходе вычислений и обновляются в результате работы метода. Выходные аргументы всегда должны сопровождаться ключевым словом **out,** обновляемые - **ref.** Что же касается входных аргументов, то, как правило, они задаются без ключевого слова, хотя иногда их полезно объявлять с параметром **ref.** Отметим, если аргумент объявлен как выходной с ключевым словом **out,** то в теле метода обязательно должен присутствовать оператор присваивания, задающий значение этому аргументу. В противном случае возникает ошибка еще на этапе компиляции.

```
/// <summary>
/// Группа перегруженных методов A()
/// первый аргумент представляет сумму кубов
/// произвольного числа оставшихся аргументов
/// Аргументы могут быть разного типа.
/// </summary>
private void A(out long p2, int p1) {
p2 = (long) Math.Pow(p1, 3);
Console.WriteLine("Метод A-1");
}
private void A(out long p2, params int[ ] p) {
p2 = 0;
for (int i = 0; i < p.Length; i++)
p2 += (long) Math.Pow(p[i], 3);
Console.WriteLine("Метод A-2");
}
private void A(out double p2, double p1) {
p2 = Math.Pow(p1, 3);
Console.WriteLine( “Метод A-3");
}
private void A(out double p2, params double[ ] p) {
p2 = 0;
for (int i = 0; i < p.Length; i++)
p2 += Math.Pow(p[i], 3);
Console.WriteLine("Метод A-4");
}
/// <summary>
/// Функция с побочным эффектом
/// </summary>
/// <param name="a"> Увеличивается на 1</param>
/// <returns> значение a на входе</returns>
private int f(ref int a) {
return a++;
}
```

- Четыре перегруженных метода с именем **A** и метод **f** будут использоваться при объяснении перегрузки и побочного эффекта. Сейчас проанализируем только их заголовки. Все методы закрыты, поскольку объявлены без модификатора доступа. Перегруженные методы с именем **A** являются процедурами, метод **f** - функцией. Все четыре перегруженных метода имеют разную сигнатуру. Хотя имена и число аргументов у всех методов одинаковы, но типы и ключевые слова, предшествующие аргументам, различны. Первый аргумент у всех четырех перегруженных методов - выходной и сопровождается ключевым словом **out,** в теле метода этому аргументу присваивается значение. Аргумент функции **f** - обновляемый, он снабжен ключевым словом **ref,** в теле функции используется его значение для получения результата функции, но и само значение аргумента изменяется в теле функции. Два метода из группы перегруженных методов используют ключевое слово **params** для своего последнего аргумента. Позже мы увидим, что при вызове этих методов указанному аргументу будет соответствовать несколько фактических аргументов, число которых может быть произвольным.

- **Тело метода**
- Синтаксически тело метода является блоком, который представляет собой последовательность операторов и описаний переменных, заключенную в фигурные скобки. Если речь идет о теле функции, то в блоке должен быть хотя бы один оператор перехода, возвращающий значение функции в форме **return (выражение).**
- Переменные, описанные в блоке, считаются локализованными в этом блоке. В записи операторов блока участвуют имена локальных переменных блока, имена полей класса и имена аргументов метода.
- Знания семантики описаний и операторов достаточно для понимания семантики блока.

Передача параметров

- Можно передавать объекты по значению, а можно по ссылке, для этого есть ключевое слово **ref**:
- static void Swap(ref int a, ref int b)
- { int t = a; a = b; b = t; }

- Если нужно просто вернуть значение из функции, а не изменить существующее значение, то нужно использовать ключевое слово **out**:

static void SolveSquareEquation(double a, double b, double c, out double x1, out double x2);

- Причем соответствующее слово должно ставиться не только в заголовке функции, но и при ее вызове
- **swap(ref aa, ref bb);**

Функции с переменным числом параметров

- В стек помещается только массив объектов, а сами объекты размещаются теперь в куче, а не на стеке Для компилятора указывается ключевое слово **params**:

```
using System;
 class Test
{
 static void F(params int[] args)
{ Console.WriteLine("# of arguments: {0}", args.Length);
for (int i = 0; i < args.Length; i++)
    { Console.WriteLine("\targs[{0}] = {1}", i, args[i]);
}
 static void Main()
{ F(); F(1); F(1, 2); F(1, 2, 3);
F(new int[] {1, 2, 3, 4}); } }
 }
```

Управление видимостью

- Есть пять типов видимости:
- **public** - функции могут вызывать кто угодно
- **protected** - функции могут вызывать только производные классы
- **private** - функции могут вызываться только из этого же класса
- **internal** - функции могут вызываться внутри пространства имен?
- **protected internal = protected + internal**
- то же самое относится ко всему, что находится внутри класса

Функции и наследование

- При наследовании можно скрыть функцию, для этого потребуется перед ней написать ключевое слово **new**. Можно переопределить, использовав ключевое слово **override**. Но переопределить можно только те функции, которые были объявлены ключевым словом **virtual**, как виртуальные (в отличие от **Java**, функции в **C#** по умолчанию невиртуальные)

Свойства

- Позволяют скрыть способ получения значения а так же реализовать дополнительную логику.

```
public string Caption
 {
get
{ return ...;
}
set
{ ... = value; ...redraw...
}
 }
```

- Если написать только функцию доступа **get**, а функцию доступа **put** пропустить, то свойство будет доступным только для чтения.

```
public static void Srednee(int[] mas)
{
    double summa=0, srednee=0;
    for (int i = 0; i < mas.Length; i++)
    {
        summa += mas[i];
    }
    srednee = summa / mas.Length;
    Console.WriteLine("Среднее арифметическое массива = " + srednee);
    Console.ReadKey();
}

public static void Srednee(float[] mas)
{
    float summa = 0, srednee = 0;
    for (int i = 0; i < mas.Length; i++)
    {
        summa += mas[i];
    }
    srednee = summa / mas.Length;
    Console.WriteLine("Среднее арифметическое массива = " + srednee);
    Console.ReadKey();
}
```

```
static void Main(string[] args)
{
    Random rnd = new Random();
    Console.Write("Выберите тип массива(1 - int; 2 - float): ");
    string s = Console.ReadLine();
    switch (s)
    {
        case "1":
            Console.WriteLine("Выбран тип массива int!");
            int[] mas = new int[10];
            for (int i = 0; i < 10; i++)
            {
                mas[i] = rnd.Next(1, 10);
                Console.Write(mas[i] + " ");
            }
            Srednee(mas);
            break;
        case "2":
            Console.WriteLine("Выбран тип массива float!");
            float[] mass = new float[10];
            for (int i = 0; i < 10; i++)
            {
                mass[i] = rnd.Next(1, 10);
                Console.Write(mass[i] + " ");
            }
            Srednee(mass);
            break;
    }
}
```