МЕТОДЫ в C#

Основные понятия

Метод – это функциональный элемент класса, который реализует вычисления или другие действия, выполняемые классом или его экземпляром (объектом). Метод представляет собой законченный фрагмент кода, к которому можно обратиться по имени. Он описывается один раз, а вызываться может многократно. Совокупность методов класса определяет, что конкретно может делать класс. Например, стандартный класс **Math** содержит методы, которые позволяют вычислять значения математических функций.

Синтаксис объявления метода:

```
[атрибуты] [спецификаторы] тип_результата имя_метода
([список_формальных_параметров])
{
        тело_метода;
        return значение;
}
```

где:

1) *атрибуты* и *спецификаторы* являются необязательными элементами в описании метода. На данном этапе атрибуты нами использоваться не будут, а из всех спецификаторов мы в обязательном порядке будем использовать спецификатор **static**, который позволяет обращаться к методу класса без создания экземпляра класса**.** Остальные спецификаторы мы рассмотрим в следующем модуле.
2) *тип_результата* определяет тип значения, возвращаемого методом. Это может быть любой тип, включая типы классов, создаваемые программистом, а также тип **void**, который говорит о том, что метод ничего не возвращает.
3) *имя_метода* будет использоваться для обращения к нему из других мест программы и должно быть корректно заданным с учетом требований, накладываемых на идентификаторы в C#.
4) *список_формальных_параметров* представляет собой последовательность пар, состоящих из типа данных и идентификатора, разделенных запятыми. Формальные параметры — это переменные, которые получают значения, передаваемые методу при вызове. Если метод не имеет параметров, то *список_параметров* остается пустым.
5) *return* – это оператор безусловного перехода, который завершает работу метода и возвращает *значение*, стоящие после оператора **return**, в точку его вызова. Тип *значения* должен соответствовать *типу__результата,* или приводиться к нему. Если метод не должен возвращать никакого значения, то указывается тип **void**, и в этом случае оператор return либо отсутствует, либо указывается без возвращаемого значения.

Рассмотрим простейший пример метода:

```
class Program
{
    static void Func()        //дополнительный метод
    {
       Console.Write("x= ");
       double x=double.Parse(Console.ReadLine());
       double y = x*x;
       Console.WriteLine("y({0})={1}", x, y );
    }
```

```
    static void Main()          //точка входа в программу
    {
      Func(); //первый вызов метода Func
      Func(); //второй вызов метода Func
    }
}
```

В данном примере в метод Func не передаются никакие значения, поэтому список параметров пуст. Кроме того, метод ничего не возвращает, поэтому тип возвращаемого значения void, и в теле метода отсутствует оператор return. В основном методе Main мы вызвали метод Func два раза. При необходимости данный метод можно будет вызвать столько раз, сколько потребуется для решения задачи.

Задания.

1. Добавьте в метод Main третий вызов метода Func.
2. Преобразуйте программу так, чтобы метод Func вызывался n раз.

Изменим исходный пример так, чтобы в него передавалось значение *x*, а сам метод возвращал значение *y*.

```
class Program
{
   static double Func( double x)       //дополнительный метод
   {
      return x*x;          //возвращаемое значение
   }

   static void Main()   //точка входа в программу
   {
     Console.Write("a=");
     double a=double.Parse(Console.ReadLine());
     Console.Write("b=");
     double b=double.Parse(Console.ReadLine());
     Console.Write("h=");
     double h=double.Parse(Console.ReadLine());
     for (double x = a; x <= b; x += h)
     {
       double y = Func(x);     //вызов метода Func
       Console.WriteLine("y({0:f1})={1:f2}", x, y);
      }
}
```

В данном примере метод Func содержит параметр x типа double. Для того, чтобы метод Func возвращал в вызывающий его метод Main значение выражения x*x (типа double), перед именем метода указывается тип возвращаемого значения – double, а в теле метода используется оператор передачи управления – *return*. Оператор *return* завершает выполнение метода и передает управление в точку его вызова.

Задания.

1. Преобразуйте программу так, чтобы метод Func возвращал значение выражения: $x^2 + x - 1$;
2. Создайте новый метод (имя придумайте самостоятельно), который будет возвращать значение выражения: $\begin{cases} x^2, & при \ \ x \le 0 \\ \sqrt{x}, & при \ x > 0 \end{cases}$

Рассмотрим другой пример:

```
class Program
{
   static int Func( int x, int y)// 1
{
   return (x>y)? x:y;
}
 static void Main()
 {
   Console.Write("a=");
   int a = int.Parse(Console.ReadLine());
   Console.Write("b=");
   int b = int.Parse(Console.ReadLine());
   Console.Write("c=");
   int c = int.Parse(Console.ReadLine());
   int max = Func(Func(a, b), c); //2 - вызовы метода Func
   Console.WriteLine("max({0}, {1}, {2})={3}", a, b, c, max);
 }
```

На этапе описания метода (строка 1) указывается, что метод Func имеет два целочисленных *формальных* параметра – x, y. На этапе вызова (строка 2) в метод передаются *фактические* параметры, которые по количеству и по типу совпадают с формальными параметрами. Если количество фактических и формальных параметров будет различным, то компилятор выдаст соответствующее сообщение об ошибке. Если параметры будут отличаться типами, то компилятор попытается выполнить неявное преобразование типов. Если неявное преобразование невозможно, то также будет сгенерирована ошибка.

Обратите внимание на то, что при вызове метода Func использовалось вложение одного вызова в другой.

Задания. Преобразуйте программу, не изменяя метод *Func*, чтобы можно было найти наибольшее значение из четырех чисел: a, b, c, d.

Таким образом, параметры используются для обмена информацией между вызывающим и вызываемым методами. В C# предусмотрено четыре типа параметров:

параметры-значения,
параметры-ссылки,
выходные параметры и
параметры, позволяющие создавать методы с переменным количеством аргументов.

При передаче параметра *по значению* метод получает копии параметров, и операторы метода работают с этими копиями. Доступа к исходным значениям параметров у метода нет, а, следовательно, нет и возможности их изменить. Все примеры, рассмотренные ранее, использовали передачу данных по значению.

Рассмотрим небольшой пример:

```
class Program
{
    static void Func(int x)
    {
        x += 10;// изменили значение параметра
        Console.WriteLine("In Func: " + x);
    }

    static void Main()
    {
       int a=10;
       Console.WriteLine("In Main: {0}", a);
       Func(a);
       Console.WriteLine("In Main: {0}", a);
    }
}
```

Результат работы программы:

```
In Main: 10
In Func: 20
In Main: 10
```

В данном примере значение формального параметра x было изменено в методе Func, но эти изменения не отразились на фактическом параметре *a* метода Main.

Замечание. *Передача параметров по значению позволяет гарантировать целостность исходных параметров только в случае, если исходные параметры имеют тип, являющийся размерным типом (value-type). Если исходный параметр имеет ссылочный тип (reference type), то в метод копируется ссылка на объект и, следовательно, можно менять значения объектов.*

При передаче параметров *по ссылке* метод получает копии адресов параметров, что позволяет осуществлять доступ к ячейкам памяти по этим адресам и изменять исходные значения параметров. Для того чтобы параметр передавался по ссылке, необходимо при описании метода перед формальным параметром и при вызове метода перед соответствующим фактическим параметром поставить спецификатор *ref*.

```
class Program
{
   static void Func(int x, ref int y)
   {
      x += 10; y += 10;  //изменение параметров
      Console.WriteLine("In Func: {0}, {1}", x, y);
   }

   static void Main()
   {
     int a=10, b=10;  // 1
     Console.WriteLine("In Main: {0}, {1}", a, b);
     Func(a, ref b);
     Console.WriteLine("In Main: {0}, {1}", a, b);
   }
}
```

Результат работы программы:

```
In Main: 10  10
In Func: 20  20
In Main: 10  20
```

В данном примере в методе Func были изменены значения формальных параметров x и y. Эти изменения не отразились на фактическом параметре a, т.к. он передавался по значению, но значение b было изменено, т.к. он передавался по ссылке.

Передача параметра по ссылке требует, чтобы аргумент был инициализирован до вызова метода (см. строку 1).

Задание. Удалите в строке 1 инициализацию переменных a и b, и попробуйте откомпилировать и запустить программу. Объясните, что произойдет.

Итак, при использовании неинициализированных фактических параметров компилятор выдаст сообщение об ошибке. Однако не всегда имеет смысл инициализировать параметр до вызова метода, например, если метод считывает значение этого параметра с клавиатуры, или из файла. В этом случае параметр следует передавать как выходной, используя спецификатор *out*..

```
class Program
{
   static void Func(int x, out int y)
   {
      x += 10; y = x;   // определение значения выходного параметра y
      Console.WriteLine("In Func: {0}, {1}", x, y);
    }

   static void Main()
   {
      int a=10, b;
      Console.WriteLine("In Main: {0}", a);
      Func(a, out b);
      Console.WriteLine("In Main: {0}, {1}", a, b);
    }
}
```

Результат работы программы:

```
In Main: 10
In Func: 20  20
In Main: 10  20
```

В данном примере в методе Func формальный параметр y и соответствующий ему фактический параметр b метода Main были помечены спецификатором out. Поэтому значение b до вызова метода Func можно было не определять, но изменение параметра повлекло за собой изменение значения параметра b.

***Замечание**. Параметры, позволяющие создавать методы с переменным количеством аргументов, основаны на использовании массивов. Изучите данный прием работы с массивами самостоятельно.*

МЕТОДЫ в C#

Перегрузка методов

Представляется разумным, чтобы методы, реализующие один и тот же алгоритм для различного количества параметров или различных типов данных, имели одно и то же имя. Использование таких методов называется *перегрузкой методов.* Компилятор определяет, какой именно метод требуется вызвать, по типу и количеству фактических параметров.

Рассмотрим следующий пример:

```
class Program
{
  //первая версия метода Max – возвращает значение наибольшей цифры
   static int Max(int a)          //заданного числа
   {
      int b = 0;
      while (a > 0)
      {
         if (a % 10 > b) b = a % 10;
         a /= 10;
       }
      return b;
   }

  //вторая версия метода Max – возвращает значение наибольшего из двух чисел
  static int Max(int a, int b)
  {
      if (a > b) return a;
      else return b;

  }

  //третья версия метода Max – возвращает значение наибольшего из трех
  //чисел
   static int Max(int a, int b, int c)
   {
      if (a > b && a > c) return a;
      else if (b > c) return b;
            else return c;
   }

   static void Main()
   {
       int a = 1283, b = 45, c = 35740;
       Console.WriteLine(Max(a));
       Console.WriteLine(Max(a, b));
       Console.WriteLine(Max(a, b, c));
   }
}
```

При вызове метода Max выбирается вариант, соответствующий типу и количеству передаваемых в метод аргументов. Если точного соответствия не найдено, выполняются неявные преобразования типов в соответствии с общими правилами. Если преобразование невозможно, выдается сообщение об ошибке. Если выбор перегруженного метода возможен более чем одним способом, то выбирается «лучший» из вариантов (вариант, содержащий меньшее количество и длину преобразований в соответствии с правилами преобразования типов). Если существует несколько вариантов, из которых невозможно выбрать подходящий, выдается сообщение об ошибке.

***Замечание**. В C# существует другой способ создания методов с переменным количеством параметров, он основывается на использовании массивов.*

Если нужно реализовать один и тот же метод для различных типов (например, наш метод Max нужно было бы реализовать для int, double и float), то совсем не обязательно записывать несколько одинаковых методов. В .NET 2.0 можно использовать сокращенную запись перегруженных методов, сообщая компилятору только общие сведения о типе, используются обобщенные типы (Generic). Они отчасти напоминают шаблоны языка C++.

При объявлении обобщенного типа мы используем некоторый формальный (не существующий тип) - его мы можем использовать как тип переменных и методов, как тип параметров и т. п. Когда же мы создаем экземпляр Generic, то мы указываем уже конкретный тип. При этом формальный тип для данного экземпляра Generic заменяется на этот конкретный тип.

Рассмотрим небольшой пример:

```
using System;
// для работы с Generic мы подключаем специальный класс
using System.Collections.Generic;
using System.Text;
namespace Example
{
   class Program
   {
      // Данный метод меняет значения двух переменных типа <T>. Тип T пока
      //не конкретизирован
      static void Swap<T>(ref T a, ref T b)
      {
         Console.WriteLine("Передаем в Swap() метод {0}",  typeof(T));
         T temp;
         temp = a;
         a = b;
         b = temp;
      }

      static void Main(string[] args)
      {
        int a = 1, b = 2;
        Console.WriteLine("Перед swap: {0}, {1}", a, b);
        Swap<int>(ref a, ref b); //передаем в Swap целый тип
        Console.WriteLine("После swap: {0}, {1}", a, b);
        Console.WriteLine();
        double x = 3.2, y = -123.27;
        Console.WriteLine("Перед swap: {0} {1}", x, y);
        Swap<string>(ref x, ref y);
        Console.WriteLine("После swap: {0} {1}", x, y);
       Console.ReadLine();
      }
   }
}
```

***Замечание**. Перегрузка методов является проявлением полиморфизма, одного из основных принципов объектно-ориентированного программирования. Программисту гораздо удобнее помнить одно имя метода и использовать его для работы с различными типами данных, а решение о том, какой вариант метода вызвать, возложить на компилятор. Этот принцип широко используется в классах библиотеки .NET. Например, в стандартном классе Console метод WriteLine перегружен 19 раз для вывода величин разных типов.*