Базовые понятия ООП: объект, его свойства и методы, класс.

Классы в языке C#

- **Класс** – описание множества схожих объектов.

  Класс определяет свойства описываемых им объектов, методы (способы) получения информации об объектах, изменения свойств и поведения объектов. Описание класса в языке C#:

- ```
  namespace MyNameSpace // Пространство имён класса
  {

      class MyClass             // Объявление класса
      {
        public static void Main()
        {
            /* Описание метода Main() класса MyClass */
        }
      }

  }
  ```

Метод Main() при запуске приложения выполняется первым.
  ```

Классы в языке C#

- Создание объекта, представляющего рассматриваемый класс, осуществляется при помощи ключевого слова **new** :

  MyClass MyObject = new MyClass() ;

- На базе одного класса может быть создано неограниченное количество объектов-представителей.

- В .NET для классов, их методов и свойств доступны следующие модификаторы доступа:
  – **public** – доступен всем;
  – **protected** – доступен только самому объекту и его потомкам;
  – **private** – доступен только самому объекту, но не его потомкам;
  – **internal** – доступен в пределах текущей сборки;
  – **protected internal** – доступен всем из текущей сборки.

- Модификаторы доступа применимы к свойствам, методам и переменным класса. Однако если требуется открыть доступ к переменной класса, её лучше превратить в свойство.

Классы в языке C#

- Модификаторы доступа также могут быть применены к классам.

  Модификатор **public** делает класс доступным для всех объектов.

  Модификатор **internal** делает класс доступным только внутри определенной сборки.

  Модификаторы **private** и **protected** могут использоваться только для вложенных классов.

- Если для переменной, свойства, метода или класса не был указан модификатор доступа, то по умолчанию для них используется модификатор **private**.

- Пример «опубликования» метода **M1()** класса **MyClass** :

```
class MyClass
{
      public static void M1()
  { /* Описание метода M1() */ }
}
```

Свойства

- Свойства – это атрибуты объектов класса.

  Свойства, как и переменные описываются в рамках определенного класса. Тем не менее, в языке C# свойства отличаются от переменных. В частности, переменные принято именовать с маленькой буквы, а свойства – с большой.

- Пример описания переменных и методов класса:

```
class MyClass
{

    int width;  // Описание private-переменной класса
    int height; // Описание private-переменной класса

    public int Width { /* Описание свойства класса */ }
    public int Height { /* Описание свойства класса */ }

}
```

Свойства

- Фигурные скобки в описании свойства позволяют задать блок действий, обеспечивающих корректную работу с данным свойством (например, задание значения свойства).
- Доступ к свойству осуществляется при помощи двух методов-аксессоров (от англ. *access* – доступ) **get** (для чтения значения) и **set** (для записи значения).

```
class MyClass
{
    int width;
    int height;

    public int Width
    {
        get { return width; } // Описание аксессора get
        set { width = value; } // Описание аксессора set
    }
}
```

Свойства

- Переменная **value** является виртуальной и имеет тот же тип, что и свойство.
- Ключевым отличием свойства от переменной является возможность ограничения операций чтения и записи его значений.

```
class MyClass
{
    int width;
    int height;

    public int Width // Свойство доступно только для чтения
    {
        get { return width; }
    }
}
```

Свойства

```
class MyClass
{
    int width;
    int height;

    public int Width   // Свойству может быть задано
                       // только положительное значение
    {
        get { return width; }

        set
        {
            if (value > 0) width = value;
        }
    }
}
```

Свойства

- Обращение к свойству объекта, представляющего описанный класс:

  ИмяОбъект.Свойство

- Пример:

```
// Создаем переменную MyObject и записываем в неё
// ссылку на созданный объект класса MyClass
MyClass MyObject = new MyClass() ;

// Устанавливаем значение свойства Width объекта MyObject
MyObject . Width = 100;

// Выводим на консоль значение свойства Width (100)
Console.WriteLine( MyObject . Width );
```

Свойства

- Аксессоры могут быть описаны с использованием модификаторов доступа.

- ```
  class MyClass
  {
      int width;
      int height;

      public int Width   // Значение свойства
                         // нельзя изменить извне
      {
          get { return width; }
          private set { width = value; }
      }
  }
  ```
  ```

Методы

- Описание метода осуществляется по следующей схеме:
  МодификаторДоступа Статичность ТипВозвращаемогоЗначения ИмяМетода(СписокПараметров)
  { *ОписаниеМетода* **}**

- ```
  class MyClass
  {
   double width;

        public void MyMethod (double width) // Описание метода
        {
              Console.WriteLine( width ); // Код метода
        }
  }
  ```
  ```

Методы

- При помощи ключевого слова **return** работу метода можно прервать в любой момент, вернув некоторое значение:

```
public double Sum (double a, double b) // Сумма двух чисел
{
 return a + b;
}
```

- Ключевое слово **return** также может быть использовано для метода, не предполагающего возврата значения (тип – **void**):

```
public void MyMethod () // Метод, не возвращающий значение
{
 return ;
}
```

Параметры методов

- По умолчанию переменные передаются в метод по значению. Это означает, что внутри себя метод создает локальные переменные с именами параметров, присваивает им указанные значения, а после работы – удаляет локальные переменные.
- Таким образом, значения самих исходных переменных по умолчанию не могут быть изменены методом.

```
static  void Product (double a,  double b) // Произведение чисел
{
 Console.WriteLine( a  *  b ); // Вывод результата на консоль

 a  +=  2; // Попытка изменить переменную a
 b  -=  2;  // Попытка изменить переменную b

 Console.WriteLine( a  *  b );  // Вывод нового результата
}
```

Параметры методов

- Проверим, изменятся ли переменные после вызова метода.

```
static  void  Main ()
{
 double  a = 3,  b = 4;  // Объявляем переменные

 Product(a,  b);  //  На консоль выводятся числа 12 и 10

 Console.WriteLine( a );  //  3
 Console.WriteLine( b );  //  4
}
```

- Примечание. Вместо имен переменных в качестве параметров метода при его вызове могут быть указаны конкретные числа.

```
Product(3,  4);  //  Ошибки не возникнет
```

Параметры методов

- Чтобы метод использовал не значение переданной переменной, а непосредственно саму переменную, следует перед указанием типа и имени параметра использовать ключевое слово **ref** (от английского *Reference* – ссылка).

```
static void Product (ref double a, ref double b)
{
  Console.WriteLine( a * b ); // Вывод результата на консоль

  a += 2; // Попытка изменить переменную a
  b -= 2; // Попытка изменить переменную b

  Console.WriteLine( a * b ); // Вывод нового результата
}
```

- **Примечание.** При вызове метода с параметром-ссылкой перед именем переменной также следует указывать **ref**. Переменная перед этим обязательно должна быть проинициализирована.

Параметры методов

- Проверим, изменятся ли переменные после вызова метода.

```
static void Main ()
{
 double a = 3, b = 4; // Объявляем переменные

 Product(ref a, ref b); // Выводятся числа 12 и 10

 Console.WriteLine( a ); // 5
 Console.WriteLine( b ); // 2
}
```

- Примечание. Если при вызове метода вместо имени переменной в качестве параметра-ссылки будет указано конкретное число, возникнет ошибка.

```
Product(3, 4); // Ошибка
```

Параметры методов

- Если переменная, указываемая в качестве параметра метода, предназначена только для возврата через неё значения бз использования внутри метода, то её можно сделать «выходной», указав перед ней ключевое слово **out**. В этом случае её не требуется инициализировать в обязательном порядке.

```
static  void Product (double a,  double b ,  out  double res)
{
 res  =  a  *  b; // Результат запишется в переменную res

  Console.WriteLine( a  *  b );  // И будет показан в консоли
}
```

- **Примечание.** При вызове метода с выходным параметром перед именем переменной также следует указывать **out**. Саму переменную инициализировать не обязательно, однако её значение обязательно должно измениться, иначе компилятор выдаст ошибку.

Параметры методов

- Проверим, изменятся ли переменные после вызова метода.

```
static  void  Main ()
{
 double  a = 3,  b = 4,  res;  // Объявляем переменные

 Product(a, b , out res);  //  Выводится число 12

 Console.WriteLine( a );  //  3
 Console.WriteLine( b );  //  4
 Console.WriteLine( res );  //  12
}
```

- Примечание. Если при вызове метода вместо имени переменной в качестве параметра-ссылки будет указано конкретное число, возникнет ошибка.

```
Product(3,  4,  12);  //  Ошибка
```

Параметры методов

- В C# имеется возможность описывать методы с произвольным количеством **однотипных** параметров. Для этого всё множество однотипных параметров указывается как один параметр-массив, перед которым используется ключевое слово **params**. Далее в методе осуществляется работа с каждым элементом массива параметров.

```
static  double Sum (params  double[] parameters)
{
 double  s = 0; //  Инициализируем переменную суммы

 foreach (double par  in parameters)  s += par; // Суммируем

 return  s;  // Возврат суммы переданных параметров
}
```

- **Пример вызова:** Console.WriteLine(**Sum**(**1, 2, 4, 6, 8**)); // 21

Параметры методов

- **Примечание.** У метода может быть только один параметр с модификатором **params**, и он должен быть указан последним в списке параметров. Без этих двух условий компилятор не смог бы определить окончание произвольного списка параметров.
- Примеры **ошибочного** использования модификатора **params**:

```
static double Sum(params string[] str, params double[] pars)
{/* Код метода */ }

static double Sum(params double[] pars, string str)
{/* Код метода */ }
```

- Правильное описание:

```
static double Sum(string str, params double[] pars)
{/* Код метода */ }
```

Перегрузка методов

- В C# имеется возможность создавать одноименные методы с различным количеством и типом параметров. В этом случае компилятор по составу параметров сможет определить, какой из одноименных методов следует использовать.

```
class Box // Описание класса объектов «Ящик»
{
    double length; double width; double height; // Размеры

    public double Length // Свойство «Длина» ящика
    { get { return length; } set { length = value; } }

    public double Width // Свойство «Ширина» ящика
    { get { return width; } set { width = value; } }

    public double Height // Свойство «Высота» ящика
    { get { return height; } set { height = value; } }
  // Здесь идёт описание методов (см. на следующем слайде)
}
```

Перегрузка методов

```
public  void  Change(double l)
{
 Length = l; //  Меняем значение свойства Length
}

public  void  Change(double l ,  double w)
{
 Length = l; //  Меняем значение свойства Length
 Width = w;  //  Меняем значение свойства Width
}

public  void  Change(double l,  double w,  double h)
{
 Length = l; //  Меняем значение свойства Length
 Width = w;  //  Меняем значение свойства Width
 Height = h; //  Меняем значение свойства Height
}
```

Перегрузка методов

```
Box b  =  new Box();      // Создаём новый объект «Ящик»
b.Length = 0;     // Инициализируем длину ящика
b.Width = 0;      // Инициализируем ширину ящика
b.Height = 0;     // Инициализируем высоту ящика

b.Change(1);      // Меняем длину ящика
Console.WriteLine(b.Length + "; " + b.Width + "; " + b.Height);
// 1;  0;  0

b.Change(2, 3);  // Меняем длину и ширину ящика
Console.WriteLine(b.Length + "; " + b.Width + "; " + b.Height);
// 2;  3;  0

b.Change(4, 5, 6);    // Инициализируем длину, ширину, высоту
Console.WriteLine(b.Length + "; " + b.Width + "; " + b.Height);
// 4;  5;  6
```

Перегрузка методов

- **Примечание.** Нельзя описывать два одноименных метода с одинаковым составом параметров (количеством и типом), поскольку компилятор не сможет их различить и выдаст ошибку.

  Следующий метод вступит в конфликт с предыдущим:

```
public void Change(double l, double w, double v)
{
  Length = l;        // Меняем длину ящика
  Width = w;         // Меняем ширину ящика
  Height = v / l / w;    // Вычисляем высоту через объём v
}
```

  Решение 1. Описать один из методов с меньшим или большим количеством параметров.

  Решение 2. Изменить тип хотя бы одного из параметров одного из методов.

  Решение 3. Комбинация предыдущих вариантов.

  Решение 4. Дать одному из методов другое имя (например, **Change_l_w_v**).

Перегрузка методов

- **Примечание.** Можно пойти на хитрость и воспользоваться приводимостью типов при модификации одного из методов.

```
public void Change(int l, double w, double v)
{
  Length = l;        // Меняем длину ящика
  Width = w;         // Меняем ширину ящика
  Height = v / l / w;    // Вычисляем высоту через объём v
}
```

В этом случае конфликта не будет, но в программе будет вызываться по метод, имеющий параметр с более узким диапазоном значений (в данном примере – метод с параметром типа **int**, считающий высоту через объём).

```
b.Change(4, 5, 6);    // Инициализируем длину, ширину, высоту
Console.WriteLine(b.Length + "; " + b.Width + "; " + b.Height);
// 4; 5; 0,3
```

Перегрузка методов

- **Примечание.** Даже в этом случае возможна неоднозначность ситуации, которая приведёт к ошибке при попытке компиляции.

```
public void Change(int l, double w, double v)
{
 Length = l;        // Меняем длину ящика
 Width = w;         // Меняем ширину ящика
 Height = v / l / w;    // Вычисляем высоту через объём v
}

public void Change(double l, int w, double h)
{
 Length = l; // Меняем значение свойства Length
 Width = w; // Меняем значение свойства Width
 Height = h; // Меняем значение свойства Height
}
```

Конструкторы

- При создании объектов класса часто возникает необходимость придавать определённые начальные значения их свойствам и переменным (то есть, инициализировать их). Для автоматизации данного процесса удобно пользоваться конструктором экземпляра класса.
- Конструктор – это специальный метод, вызываемый при создании экземпляра класса. Конструктор ничего не возвращает, а только производит подготовительные операции при создании объекта класса (при этом ключевое слово **void** указывать не требуется).
- Имя конструктора всегда совпадает с именем класса:
-

```
class MyClass
{
 double width;
      public   MyClass ()  // Обращение к конструктору класса
      {
           width  =  0;  // Инициализация переменной width
      }
}
```

Конструкторы

- В приведенном примере модификатор доступа **public** позволяет обращаться к конструктору данного класса в других классах. Чтобы отменить такую возможность, следует использовать модификатор доступа **private** (или же просто не указывать его):
-
```
class MyClass
{
 double width;
      private   MyClass ()  // Или просто MyClass()
      {
           width  =  0;  // Инициализация переменной width
      }
}
```
- **Замечание.** Если для работы с переменными в классе были созданы свойства, то лучше изменять значение свойств, а не самих переменных. В этом случае при работе со свойством будет запущен обработчик, проверяющий корректность значения, заносимого в переменную через данное свойство.

Конструкторы

- Конструктор, как и любой метод, может использовать параметры. При этом бывают ситуации, когда имя переменной класса совпадает с именем одного из параметров. Например:
-
```
class MyClass
{
 double width;
      public  MyClass (double width)
      {
           width = width; // Совпадение имён
                          // переменной и параметра
      }
}
```
- **Вопрос.** Как в этом случае указать компилятору, откуда мы берём значение и куда мы его хотим присвоить?

Конструкторы

- Для указания ссылки на текущий объект следует использовать ключевое слово **this**. Например:
-
```
class MyClass
{
 double width;
      public  MyClass (double width)
      {
            this.width = width;     // Теперь имена переменной
                                    // и параметра различимы
      }
}
```

- **Замечание.** Вместо **this.width** нельзя писать **MyClass.width**, поскольку в этом случае мы обращаемся не к конкретному объекту, а к классу, для которого не выделяется память под хранение значений переменных.

Конструкторы

- Чтобы модифицировать конструктор по умолчанию, нужно указать его без параметров. Однако можно создать дополнительный конструктор для определённого набора параметров. Таких конструкторов может быть создано сколько угодно для различных ситуаций.
- Также имеется возможность использовать в конструкторе по умолчанию один из дополнительных конструкторов. Для указания ссылки на текущий объект следует использовать ключевое слово **this**. Например:
-

```
class MyClass
{
 double width;
 // Описание конструктора MyClass(double width)
      public   MyClass()  :  this.MyClass (0)
      {
            //  this.width  =  0;  теперь писать не нужно
      }
}
```

Статичность методов

- В отличие от объектов классы не обладают собственной памятью, а представляют собой лишь шаблонное описание бесконечного множества схожих объектов. Как следствие, класс в принципе не способен осуществлять вызов методов объектов.
- Статичный метод может использовать только те внешние переменные, которые сами объявлены как статичные. Это связано с тем, что статичные методы и переменные привязываются к классу, а не к его объектам. Таким образом, на базе рассматриваемого класса может быть создано сколь угодно много объектов, но статичная переменная класса для всех этих объектов всегда будет одна.
- Для придания статичности методу или переменной используется ключевое слово **static** перед их описанием:

  static *ТипПеременной ИмяПеременной* ;

  static *ТипВозвращаемогоЗначения ИмяМетода (СписокПараметров)*;

- Пример:

```
static int MyVar; // Статичная переменная MyVar
static int MyMethod (int MyVar); // Статичный метод MyMethod
```

Статичность методов

- Отмеченная особенность статичных переменных может быть использована, например, для подсчёта количества объектов, созданных на базе рассматриваемого класса:

```
class MyClass
{
    static int ObjectNumber; // Переменная счётчика

    public int GetObjectNumber()
    {
        return ObjectNumber; // Получаем количество
    }

    public MyClass()
    {
        ObjectNumber++; // Увеличиваем счётчик
    }
}
```

Статичность методов

- Проверка работы счётчика:

```
// Создаем первый объект класса MyClass
MyClass MyObject1 = new MyClass() ;

// Получаем значение счётчика
Console.WriteLine( MyObject1 . GetObjectNumber() ); // 1

// Создаем первый объект класса MyClass
MyClass MyObject2 = new MyClass() ;

// Получаем значение счётчика
Console.WriteLine( MyObject2 . GetObjectNumber() ); // 2
Console.WriteLine( MyObject1 . GetObjectNumber() ); // 2
```

Деструкторы

- Платформа .NET по мере необходимости осуществляет автоматическое освобождение выделенной в процессе работы памяти. Определение ненужности объекта в текущий момент работы программы определяется по количеству оставшихся ссылок на него. Как только количество ссылок станет равно нулю, объект добавляется в список подлежащих уничтожению. Впоследствии память из-под такого объекта будет освобождена так называемым сборщиком мусора, и объект прекратит свое существование. Для принудительного начала сборки мусора нужно обратиться к методу Collect() класса GC, соответствующего сборщику мусора:
- **GC.Collect();**

- Метод, вызываемый в ответ на уничтожение объекта, называется деструктором. Как и у конструктора, его имя совпадает с именем класса, но перед ним добавляется символ тильды:
- ```
  class MyClass
  {
      ~MyClass () { } // Обращение к деструктору класса
  }
  ```
  ```

Статичность методов

- Деструкторы могут быть использованы совместно с конструкторами, например, для подсчёта количества объектов данного класса, существующих в текущий момент работы программы:

- ```
  class MyClass
  {
      // Содержимое класса, описанное на предыдущем слайде

      public ~MyClass()
      {
          ObjectNumber--; // Уменьшаем счётчик
      }
  }
  ```
  ```