КЛАССЫ

Определение класса

Класс – это шаблон, который определяет форму объекта. Он задает данные и код, который оперирует этими данными. ЯП C# использует спецификацию класса для создания объекта.

Таким образом класс – это множество намерений (планов), определяющих, как должен быть построен объект. Класс – это логическая абстракция, о ее реализации нет смысла говорить до тех пор, пока не создан объект класса и в памяти не появилось его физическое представление.

Форма описания класса

[атрибуты] [спецификаторы] class имя_класса [: предки]

{

тело класса

}

КОМПОНЕНТЫ КЛАССА

- индексаторы;
- свойства;
- статические переменные;
- типы;
- константы;
- конструкторы;
- деструкторы;
- события;
- операторы.

Если класс не содержит методов, его называют классом данных.

Компоненты класса

- Константы хранят неизменяемые значения, связанные с классом
- Поля содержат данные класса
- Методы реализуют действия, выполняемые классом или экземпляром
- Свойства определяют характеристики класса в совокупности со способами их задания и получения (методами записи и чтения)
- Конструкторы выполняют действия по инициализации экземпляров или класса в целом
- Деструкторы определяют действия, выполняемые перед тем как объект будет уничтожен

Компоненты класса

- Индексаторы обеспечивают возможность доступа к элементам класса по их порядковому номеру
- Операции задают действия с объектами с помощью знаков операций
- События, на которые может реагировать объект, определяют уведомления, которые может генерировать класс
- Типы – внутренние по отношению к классу типы данных

```
class MyClass
{
    int value1;
    //Переменная целого типа.
    //При создании класса станет равной 0

    const double value2 = 23.3435;
    //Константа дробного типа.

    readonly short value3 = 45;
    //Переменная "Только для чтения"

    string str1 = "123456";
    //Строка, объявляется одновременно с инициализацией
}
```

Модификаторы доступа

public	Элемент доступен всем.
private	Элемент доступен только в том типе, в котором он определен.
protected	Элемент доступен в типе, в котором он определен, и в его потомках.
internal	Элемент доступен только в текущей сборке, в других сборках – не виден.
protected internal	Работает как **`protected`** и как **`internal`** .
abstract	Абстрактный класс, свойство, метод.
sealed	Бесплодный класс, отключение наследования.
static	Статический метод или переменная.
new	Для вложенных классов либо для скрытия унаследованных переменных.

По умолчанию - private.

Статические переменные и методы

Если поле или метод объявлены как static (т.е. имеет одно и то же значение для всех экземпляров класса), к нему можно получить доступ до создания объекта этого класса и без ссылки на объект.

Простейший пример – функция Main().

```
namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
```

Переменные, объявленные как static, являются по сути глобальными переменными. Все экземпляры класса используют одну и ту же static-переменную. Она инициализируется при загрузке класса, или по умолчанию - нулем, false, null.

Ограничения статических методов

1. Не имеют ссылки (указателя) this.
2. Могут напрямую вызвать только другие static-методы.
3. Могут получать прямой доступ только к static-переменным.
4. Статические поля класса и константы существуют в единственном экземпляре для всех объектов класса. Статическими могут быть конструкторы. Они обычно используются для инициализации атрибутов, которые применяются к классу в целом, а не к конкретному его экземпляру.

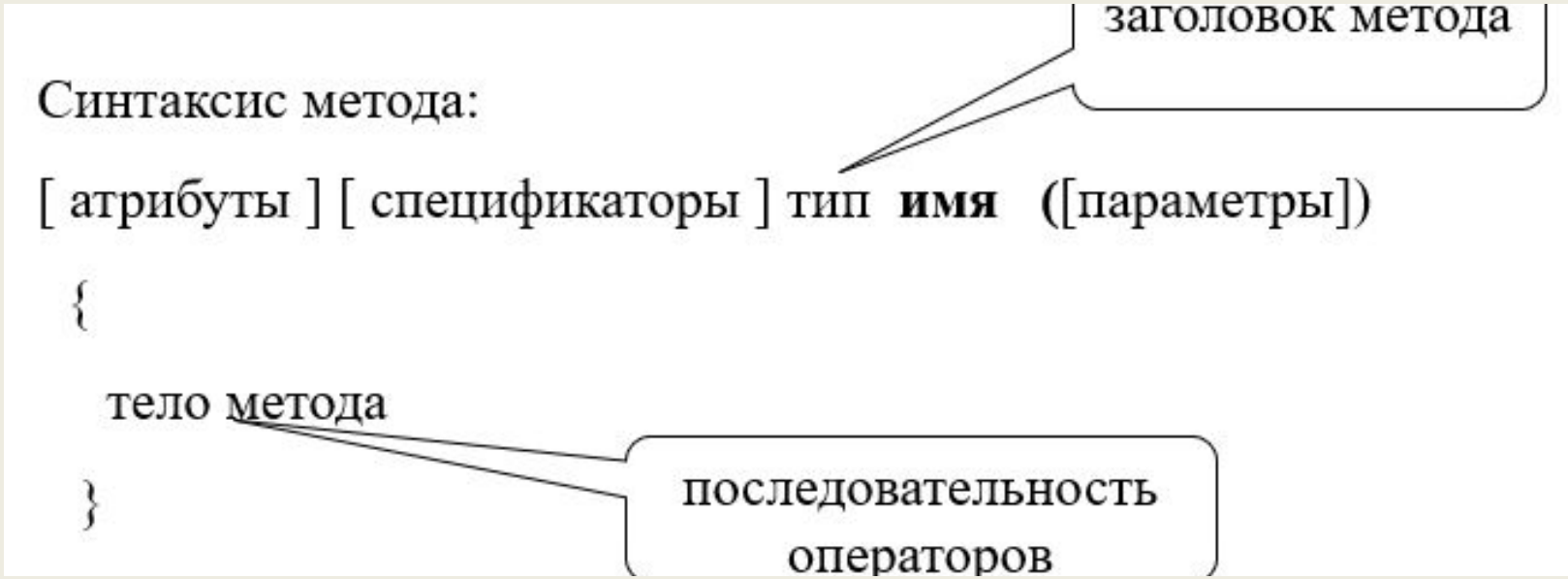
Для вызова нестатическоrо метода необходим экземпляр класса. Для вызова статического метода требуется имя класса, а не экземпляр.

```
//класс со статической и нестатической переменной
// статическим и нестатическим методом
class Stat
{
    int d = 3;
    public static int val = 100;

    //public void nonStatic() { Console.WriteLine("Non static method"); }
    //все правильно

    public static int valDiv2() { return val / 2; }
    //все правильно
    //public static int valDiv2() { return val / d; }
    //!ОШИБКА! нужна ссылка на объект

    //public static int valDiv2() { nonStatic(); return val / 2; }
    //!ОШИБКА! нужна ссылка на объект
}
```

```
static void Main(string[] args)
{
    Console.WriteLine("Начальное значение Stat.val " + Stat.val);
    Stat.val = 8;
    Console.WriteLine("Значение Stat.val " + Stat.val);
    Console.WriteLine("Начальное значение Stat.valDiv2 " + Stat.valDiv2());
    Console.ReadKey();
} *
```

Метод

- оформленный особым образом поименованный фрагмент кода, который реализует вычисления или другие действия, выполняемые классом или объектом
- Метод описывается один раз, а вызываться может необходимое количество раз.



Спецификаторы метода

Спецификаторы метода:

public	доступ не ограничен
protected	доступ только из данного и производных классов
internal	доступ только из данной сборки
private	доступ только из данного класса (по умолчанию)
static	одно поле для всех экземпляров класса

Для работы со статическими данными класса используются статические методы (static), для работы с данными экземпляра – методы экземпляра (просто методы).

Статические методы можно вызвать, не создавая объект.

Чаще всего применяется спецификатор доступа к методам public.

Возвращаемое значение

Для передачи значения выражения в качестве результата работы метода используется оператор

return выражение;

Если метод не возвращает никакого значения, в заголовке указывается тип **void,** а оператор **return** отсутствует.

Пример 1:

```
class Primer2
   {  string s;
      public string vvods()
     {
        s = Console.ReadLine();
        return s;
      }
```

Пример 2:

```
public void vyvods()
        { Console.WriteLine(s); }
```

Параметры метода

Параметры в заголовке используются для обмена информацией с методом и определяют множество значений аргументов, которые можно передавать в метод.

Для каждого параметра указывается тип.

Обращение к статическому методу класса:

имя класса. имя метода([аргументы])

Обращение к нестатическому методу класса:

имя объекта. имя метода([аргументы])

При вызове метода с параметрами количество аргументов должно совпадать с количеством параметров в заголовке метода. Кроме того должно существовать неявное преобразование типа аргумента к типу соответствующего параметра.

Параметры метода

В C# предусмотрены след. виды параметров:

- параметры-значения
- параметры-ссылки
- выходные параметры
- параметры-массивы

Параметр-значение

При описании параметра-значения в заголовке метода указывается только тип.
Параметр- значение представляет собой локальную переменную, которая получает в качестве своего значения *копию* значения аргумента.

При вызове метода в качестве соответствующего аргумента на месте параметра-значения может находиться выражение, а в том числе переменная и константа.

```
static int maximum(int x, int y)
{
    if (x > y) return x;
     else return y;
}
```

Параметр-ссылка

Если в методе требуется изменить значение передаваемых в качестве параметров величин, используют *параметры-ссылки*.

При описании параметра-ссылки в заголовке метода перед указанием типа помещают ключевое слово **ref**

При вызове метода в область параметров копируется не значение аргумента, а его адрес, т.е. метод работает непосредственно с переменной из вызывающей функции и может ее изменить.

При вызове метода в качестве аргумента на месте параметра-ссылки может находиться **только ссылка на инициализированную переменную точно того же типа со словом ref перед аргументом.**

```
static void Udvoenie(ref int x, ref float y)
    { x = 2 * x;  y *= 2;
    }
```

Выходные параметры

Если нет необходимости инициализировать переменную-аргумент до вызова метода, можно использовать выходные параметры.

При описании выходного параметра в заголовке метода перед указанием типа помещают ключевое слово **out.**

Выходному параметру **обязательно** должно быть присвоено значение внутри метода, а в вызывающем коде переменную достаточно описать.

При вызове метода перед соответствующим аргументом тоже указывается слово **out.**

```
static void Vvod(out int x, out float y)
    {
      string s;
      Console.WriteLine("Введите целое число");
      s = Console.ReadLine(); x = Convert.ToInt32(s);
      Console.WriteLine("Введите вещественное число");
      s = Console.ReadLine(); y = Convert.ToSingle(s);
    }
```

Конструкторы и деструкторы в C#

В классе возможно объявить любое количество конструкторов с разной сигнатурой (различными количеством и типом принимаемых параметров).

Если в классе не объявлено ни одного конструктора, создается конструктор по умолчанию, не принимающий никаких параметров. Однако, если в классе объявлен хоть один конструктор с параметрами, то конструктор без параметров, если он нужен, необходимо дописывать самостоятельно.

```
class Point
{
    private int x;
    private int y;

    public Point() {}

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

Конструкторы

Для создания объекта в C# можно использовать конструкцию:

```
Point point1 = new Point(10, 20);
Point point2 = new Point();
//Point point3;
```

Она объявляет переменную-объект, которой присваивается ссылка на физический объект. Оператор new динамически выделяет память для объекта и возвращает ссылку на него.

В C# так поступают со всеми объектами.

Конструкторы - 2

```
MyClass MyObject = new MyClass();
```

В этом случае MyObject – переменная, которая может ссылаться на объект, но не сам объект. Поэтому классы в C# называют ссылочными типами.

Память, выделенную с помощью оператора new, необходимо восстанавливать (освобождать).

Система сбора мусора

Автоматически восполняет память для повторного использования, действуя незаметно и без вмешательства программиста.

Можно также определить метод, который должен вызываться непосредственно перед тем, как объект будет окончательно разрушен системой сборки мусора. Все деструкторы неявно вызываются перед завершением работы программы.

Перегрузка операторов

Название оператора	Оператор	Возможности перегрузки
Унарные	+, -, !, ~, ++, --	Можно
Бинарные	+, -, *, /, %, &, |, ^, <<, >>	Можно
Сравнения	==, !=, <, >, <=, >=	Можно, но только парами
Условные логические	&&, ||	Нельзя, но они вычисляются с помощью & и |
Индексирования	[]	Нельзя, но можно определить индексатор
Приведения типа	()	Нельзя, но можно определить новые операторы преобразования explicit (явно) и implicit(неявно)
Составные операторы присваивания	+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Нельзя, но += вычисляется с помощью +, который допускает перегрузку
Иные	= . ?: -> sizeof new is typeof	нельзя

ИНДЕКСАТОРЫ, АКСЕССОРЫ И СВОЙСТВА

Свойства (property)

Свойства - способ управления доступом к данным экземпляра класса.
Формат записи свойства:

```
тип <Имя_Свойства>
{
    get
    {
    //аксессор чтения данных (получения)
    }
    set
    {
    //аксессор записи данных (установки)
    }
}
```

Свойство ничего не хранит, оно выступает в роли посредника между внешним кодом и переменной

Аксессоры get и set

Аксессоры (accessor) похожи на методы, за исключением того, что в них отсутствует тип возвращаемого значения и параметры.

Аксессор set автоматически принимает параметр с именем value, который содержит значение, присваиваемое свойству.

```
set {name= value};
```

Аксессор **get** возвращает значение поля:

```
get {return name};
```

Особенности свойств

1. Свойства нельзя передавать в качестве параметра.
2. Свойства нельзя перегружать.
3. Свойство не определяет область хранения поля, а только управляет доступом к нему.
4. Можно определять свойства только для чтения или только для записи.

Пример

```
//класс для работы со свойствами
class SimpProp
{
    int x;//это поле управляется свойством.
    //Поле закрытое, доступ только через свойство myprop
    public SimpProp() { x = 0; }//конструктор
    //свойство позволяет присваивать переменной
    //только положительные числа
    //предназначено для чтения и записи
    public int myprop
    {
        get//для чтения
        {
            return x;//возвращаем поле
        }
        set//для записи
        {
            if (value >= 0) x = value;//иначе - остается прежним
        }
    }
}
```

Пример. Продолжение

```
namespace ConsoleApplication2
{
    class Program
    {
        static void Main(string[] args)
        {
            SimpProp ob = new SimpProp();//объект - экземпляр класса
            Console.WriteLine("Исходное значение поля: "+ob.myprop);// 0
            ob.myprop = 100;//доступ к полю закрыт, присваиваем через свойство
            Console.WriteLine("Значение поля: " + ob.myprop);//100
            ob.myprop = -10;//попытка присвоить отрицательное значение
            Console.WriteLine("Значение поля: " + ob.myprop);//100
            Console.ReadKey();
        }
    }
}
```

Индексаторы (indexer)

Индексаторы обеспечивают механизм, с помощью которого к объектам можно получить доступ по индексу.
Одномерный индексатор имеет вид:

```
возвращаемый_тип this [Тип параметр1, ...]
{
    get { ... }
    set { ... }
}
```

В индексаторах можно опускать блок get или set

Особенности

- Индексатор не имеет названия. Вместо него указывается ключевое слово this, после которого в квадратных скобках идут параметры.
- Индексатор должен иметь как минимум один параметр.
- Индексаторы могут быть перегружены.
- Можно использовать индексаторы для многомерных массивов.

Пример 2

```
class Vector
{
    int[] points = new int[100];

    public int this[int i]
    {
        get
        {
            if (i < points.Length && i >= 0)
                return points[i];
            return 0;
        }
        set
        {
            if (i < points.Length && i >= 0)
                points[i] = value;
        }
    }
}
```

Пример 2

```
static void Main(string[] args)
{
    Vector vec = new Vector();
    vec[0] = 10;
    vec[1] = 25;
    vec[-2] = 17;   //Неверный индекс, аварийного завершения не произойдет
    Console.WriteLine(vec[0].ToString());
    Console.WriteLine(vec[1].ToString());
    Console.WriteLine(vec[2].ToString());
    Console.WriteLine(vec[-2].ToString());
}
```

Пример 2

```
class MyArray
{
    //поля
    int[] a;//массив
    public int Length;//размерность
    public bool errflag;//флаг ошибки
    //методы
    //конструктор
    public MyArray(int size)
    {
        a = new int[size];
        Length = size;
    }
    //проверка границ
    private bool ok(int index)
    {
        if (index >= 0 & index < Length) return true;
        return false;
    }
```

```
//индексатор
public int this[int index]
{
    get //аксессор считывания
    {
        if (ok(index))//если в границах
        {
            errflag = false;
            return a[index];//возвращаем значение
        }
        else
        {
            errflag = true;
            return 0;
        }
    }
    set//аксессор установки значений
    {
        if (ok(index))//если в границах
        {//устанавливаем значение
            a[index] = value; //атоматически устанавливаемый параметр
                              //- значение для записи
            errflag = false;
        }
        else errflag = true;
    }
}*
```

```
MyArray P = new MyArray(5);//объект на 5 элементов
int x;
Console.WriteLine("Работа с уведомлением об ошибках");
Console.WriteLine("Запись");
for (int i = 0; i < P.Length * 2; ++i){
    P[i] = i * 10;
    if (P.errflag)
        Console.WriteLine("P[" + i + "] вне границ");
}
Console.WriteLine("Чтение");
for (int i = 0; i < P.Length * 2; ++i)
{
    x = P[i];
    if (!P.errflag) Console.WriteLine(x + " ");
    else Console.WriteLine("P[" + i + "] вне границ");
}
```