

Понятие класса, объекта

Введение в ООП

ТРАДИЦИОННАЯ

АЛГОРИТМ
ОБЪЕКТ
(ФУНКЦИЯ)

ДАННЫЕ

1

2

Не ориентированный на объекты подход



процедуры



данные на входе



вызовы других
процедур

```
Procedure Вскипятить_чайник  
begin
```

```
    Зажечь плиту;
```

```
    Взять чайник;
```

```
    Налить в чайник воды;
```

```
    Поставить на плиту;
```

```
    Подождать 5 минут;
```

```
end
```

```
begin
```

```
    if Чайник не пуст then
```

```
        Вылить из чайника воду;
```

```
        Вскипятить_чайник;
```

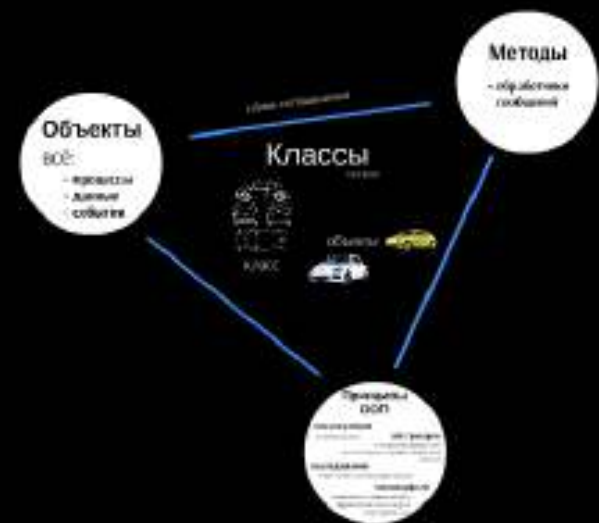
```
end
```

Объектно-ориентированный подход

```
class Плита {  
    //boolean  
    Горит Ли Конфорка? (конфорка)  
  
    Зажечь Конфорку (конфорка);  
  
    Потушить Конфорку (конфорка);  
  
    Установить Уровень Нагрева (конфорка, уровень);  
}
```

```
class Чайник {  
    // boolean  
    Пустой ли Чайник();  
  
    // boolean  
    В Процессе Нагрева();  
  
    // Возвращает boolean (удалось или нет)  
    Поставить На Плиту(плита, конфорка);  
}
```

```
плита.Зажечь Конфорку(конфорка)  
плита.Установить Уровень Нагрева(конфорка, 6)  
чайник.Поставить На Плиту(плита, конфорка)
```



ОО подход

Данные+Логика



Объект

АВТОНОМНЫЙ модуль со своим состоянием и поведением

- 1) обладает состоянием
- 2) имеет четкие границы
- 3) имеет набор действий

ИТОГО

- 1) Все является **объектом**
- 2) **Программа** = группа объектов, которые общаются между собой
- 3) Каждый объект имеет **состояние**
- 4) Каждый объект имеет свой **тип**

Объектно-ориентированное программирование

- ▶ это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования

Принципы объектно-ориентированного программирования

- ▶ Наследование (Inheritance);
- ▶ Инкапсуляция (Encapsulation);
- ▶ Полиморфизм (Polymorphism).
- ▶ Абстракция данных

Инкапсуляция (пакетирование)

- ▶ механизм, связывающий вместе данные и код, обрабатывающий эти данные, и сохраняющий их от внешнего воздействия и ошибочного использования

- 1) Никто не знает что внутри
- 2) Никто не может менять данные снаружи



Свойства инкапсуляции

- ▶ Совместное хранение данных и функций
- ▶ Соккрытие внутренней информации от пользователя
- ▶ Изоляция пользователя от особенностей реализации

Абстрактные типы данных

Абстракция подразумевает разделение и независимое рассмотрение **интерфейса** и **реализации**

- ▶ абстракция - уровень описания/представления модели чего либо

Наследование

► процесс, благодаря которому один объект может наследовать (приобретать) свойства от другого объекта.

► *иерархии классов*

Student -> GradStudent



Полиморфизм



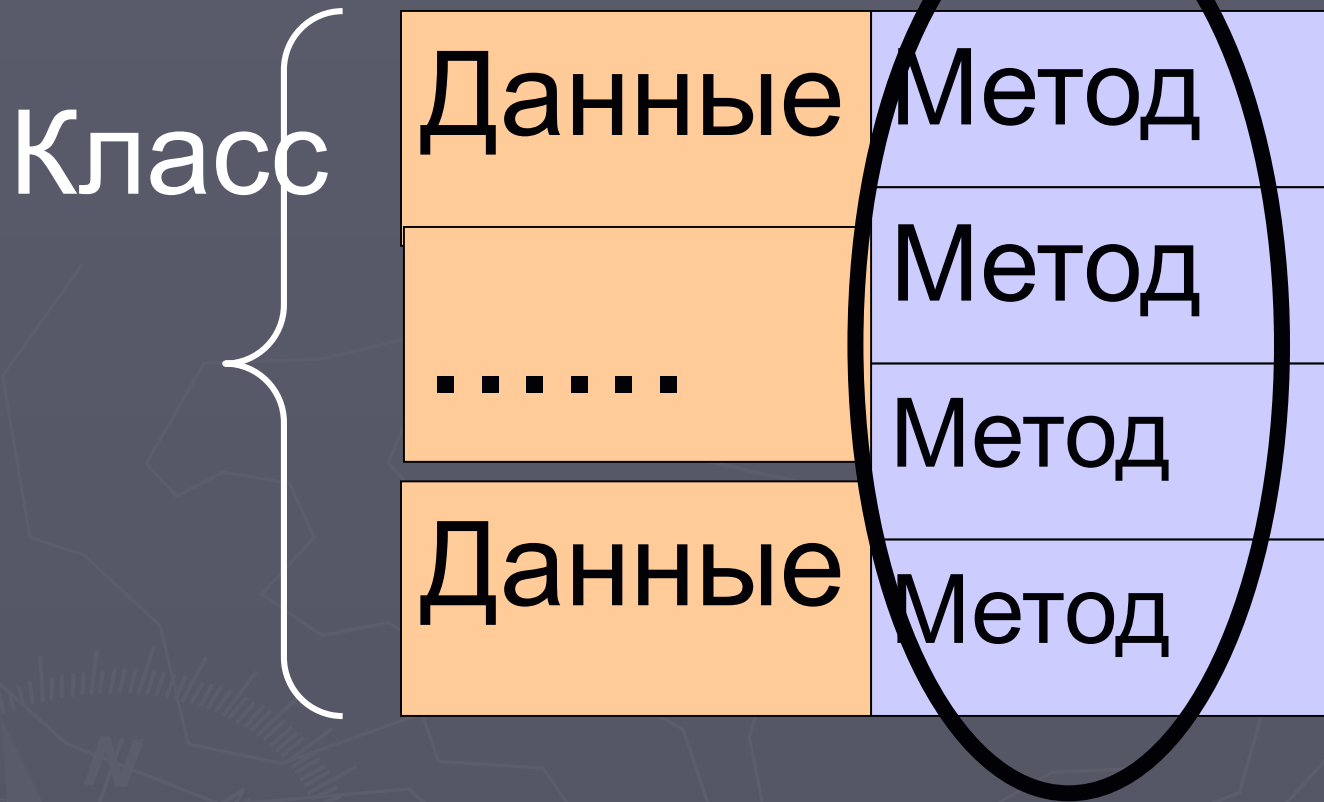
- ▶ -способность вызывать метод потомка через экземпляр предка
- ▶ **-один интерфейс — множество методов**
- ▶ Поддержка полиморфизма осуществляется через *виртуальные* функции, механизм перегрузки функций и операторов, а также обобщения

ОБЪЕКТ

структурированная
переменная, содержащая
всю информацию о
некотором физическом
предмете или реализуемом в
программе понятии.

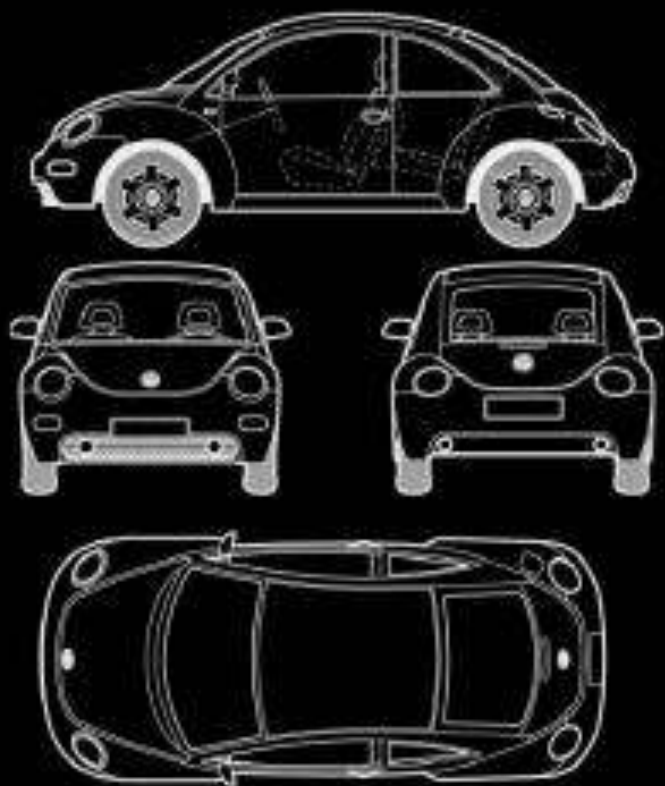
КЛАСС

множества таких
объектов и
выполняемых над
ними действий.



► *Класс* – это некоторое абстрактное понятие - шаблон, по которому определяется форма объекта

► *Объект* – это физическая реализация класса(шаблона).



класс

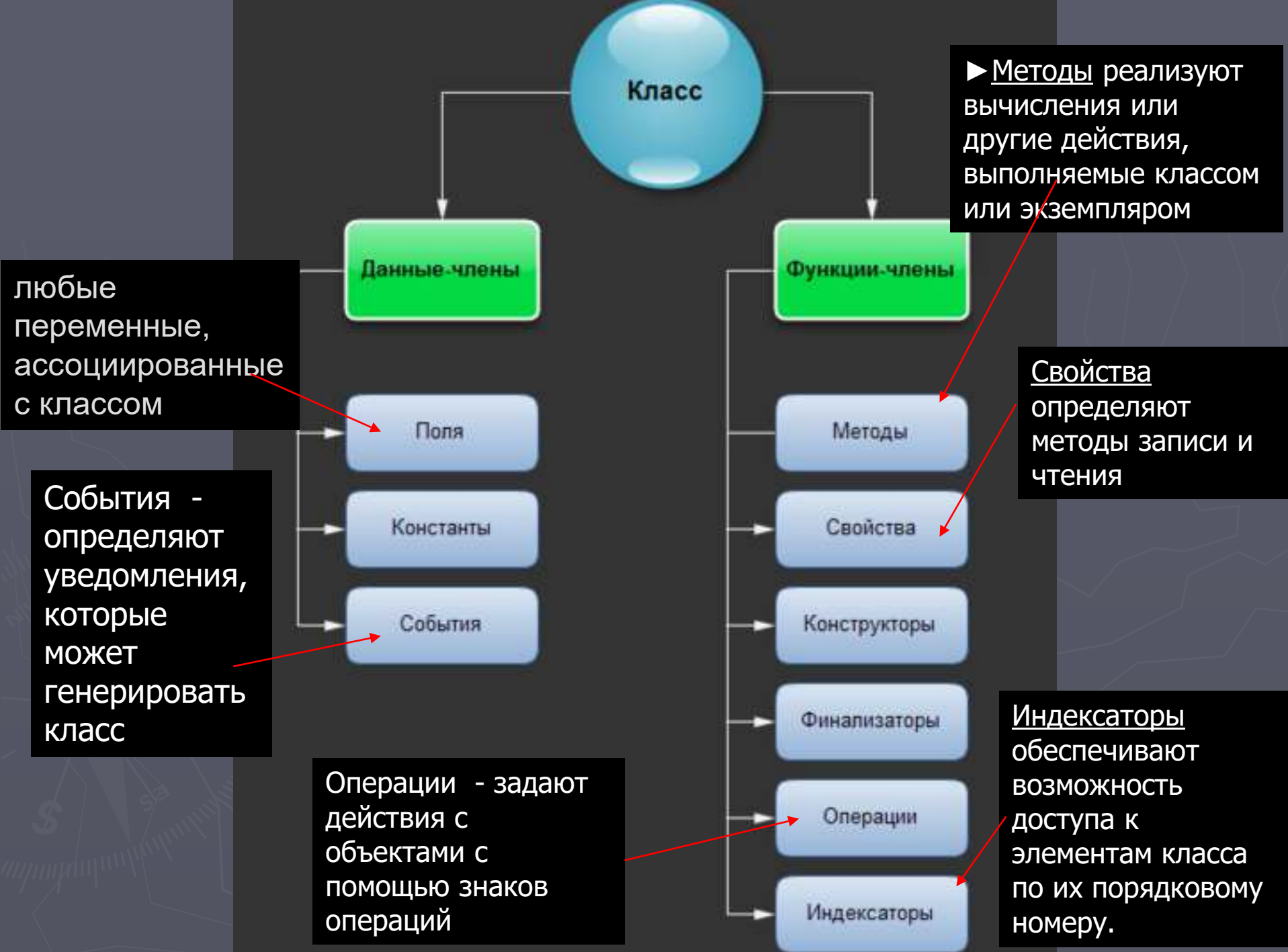


объекты



Класс в C# - *user defined type, UDT*

[атрибуты] [спецификаторы]
class имякласса [: предок]
{
 тело-класса
}



```
public class Student
```

```
{
```

```
    public string name;
```

```
    public string secondName;
```

```
    public int course;
```

```
    public void Info()
```

```
{
```

```
        Console.WriteLine($"Студент {name} учится  
                             на {course} ");
```

```
}
```

```
}
```

```
class Program
```

```
{
```

```
    static void Main(string[] args)
```

```
{
```

```
        Student Olga = new Student();
```

```
        Olga.Info();
```

```
}}
```

Определение
класса

Спецификатор
доступа

Создание
объекта

Обращение –
имя_Объекта.Имя_члена

Студент учится на 0

Константы

```
const int CC =100;  
// значение не может изменено  
readonly int FC;
```

- 1) компилятор сохраняет значение константы в метаданных модуля → константы можно определять только для таких типов, которые компилятор считает примитивными (или не примитивный но тогда = null)
- 2) константы считаются не явно статическими, всегда связаны с типом, а не с экземпляром типа
- 3) нельзя получать адрес константы и передавать ее по ссылке
- 4) объявить можем один раз
- 5) к моменту компиляции они должны быть определены.

```
class Program
{
    private static void Main(string[] args)
    {
        Point end = new Point();
        end.y = 45;
    }
}

class Point
{
    public int x;
    public const int y;
}
```

☒ (константа) `int Point.y`

Требуется указать значение поля const.

```
public class Student
```

```
{
```

```
    public string name;
```

```
    public string secondName = "NoName";
```

```
    public int course;
```

```
    private const string UO = "БГТУ";
```

Инициализация
экземплярного
поля (inline)

константа

```
    public void Info()
```

```
{
```

```
        Console.WriteLine($"Студент {name} учится на  
                             {course} курсе в {UO}");
```

```
}
```

```
}
```

```
class Program
```

```
{
```

```
    static void Main(string[] args)
```

```
{
```

```
        Student Olga = new Student();
```

```
        Olga.name = "Ольга";
```

```
        Olga.course = 2;
```

```
        Olga.Info();}}
```

Студент Ольга учится на 2 курсе в БГТУ

Поля для чтения

readonly - инициализация времени испол.

- 1) Запись в поле разрешается при объявлении или в коде конструктора
- 2) Инициализировать или изменять их значение в других местах нельзя, можно только считывать их значение.

```
class Point
{
    public int x;
    public readonly int y = 0; // можно так инициализировать
    public Point (int _y)
    {
        y = _y; //может быть инициализировано
    } //или изменено в конструкторе после компиляции
    public void ChangeY(int _y)
    {
        y = _y; // нельзя
    }
}
```

Видимость типа

- ▶ может быть открытым (public) или внутренним (internal).

По умолчанию для класса

```
// Открытый тип доступен из любой сборки  
public class Машина { }
```

```
// Внутренний тип доступен только из собственной сборки  
internal class Колесо { }
```

```
// Это внутренний тип, так как модификатор доступ  
а не указан явно  
class Двигатель { }
```


Доступ к членам типов

- ▶ **public** - доступ не ограничен — все методы во всех сборках
- ▶ **private** - по умолчанию для членов класса (используется для вложенных классов). Доступен только методам в определяющем типе и вложенных в него типах
- ▶ **protected** - (используется для вложенных классов) Доступен только методам в определяющем типе (и вложенных в него типах) или в одном из его производных типов независимо от сборки
- ▶ **internal** - доступ только из данной сборки

Модификаторы определяют, на какие члены можно ссылаться из кода

```
public class Student
```

```
{
```

```
    private string name;
```

```
    string secondName = "NoName";
```

```
    protected int course;
```

```
    private const string UO = "БГТУ";
```

► Все поля класса закрытые -
спецификатор доступа private

Доступ к состоянию объекта —
верный путь к непредсказуемому
поведению и проблемам с
безопасностью.

```
    public void Info()
```

```
{
```

```
        Console.WriteLine($"Студент {name} {secondName}  
                           учится на {course} курсе в {UO}");
```

```
}
```

```
}
```

```
class Program
```

```
{
```

```
    static void Main(string[] args)
```

```
{
```

```
        Student Olga = new Student();
```

```
        Olga.name = "Ольга";
```

```
        Olga.course = 2;
```

```
        Olga.Info();}}
```

► Чаще всего для методов задается
спецификатор доступа public

Недоступны из-за уровня
защиты

```
public class TestAccess
{
    ✗ int age; // == private int age;
    ✗ private int birthday;
        // доступно только из текущего класса
    ✗ protected int date;
        // доступно из текущего класса и производных классов
    internal int sum;
        // доступно в любом месте программы
    protected internal int email;
        // доступно в любом месте программы
        //и из классов-наследников
    public int adres;
        // доступно в любом месте программы,
        //а также для других программ и сборок (dll)
}
```

Инкапсуляция - скрывание некоторых моментов реализации класса от других частей программы.

Перегрузка методов

- ▶ один и тот же метод, но с разным набором параметров
- ▶ *позволяет обращаться к связанным методам посредством одного, общего для всех имени.*
- ▶ **никакие два метода внутри одного и того же класса не должны иметь одинаковую сигнатуру**

сигнатура (signature) = имя метода + список его параметров (не включает тип значения, возвращаемого методом, не включает params-параметр)

```
// -----Перегрузка методов-----  
    // Возвращает наибольшее из двух целых:  
class Test{  
    public int max( int a, int b ) {return 1;}  
    // Возвращает наибольшее из трех целых:  
    public int max(int a, int b, int c) { return 2; }  
    // Возвращает наибольшее из первого параметра и длины второго:  
    public int max(int a, string b) { return 3; }  
    // Возвращает наибольшее из второго параметра и длины первого:  
    public int max(string b, int a) { return 4; }  
    public int max(int a, ref int b) { return 5; }  
}  
public static void Main()  
{  
    Test q = new Test();  
    Console.WriteLine(q.max(1, 2));  
    Console.WriteLine(q.max(1, 2, 3));  
    Console.WriteLine(q.max(1, "222"));  
    Console.WriteLine(q.max("123", 2));  
  
}
```

Конструкторы

Конструкторы — это специальные методы, позволяющие корректно инициализировать новый экземпляр типа.

Создание экземпляра объекта ссылочного типа

- 1) выделяется память для полей данных экземпляра
- 2) инициализируются служебные поля
- 3) вызывается конструктор экземпляра, устанавливающий исходное состояние нового объекта
- 4) память всегда обнуляется до вызова конструктора экземпляра типа. Любые поля, не задаваемые конструктором явно, гарантированно содержат 0 или null.


Свойства конструкторов

- ▶ 1) не имеет возвращаемого значения
- ▶ 2) имя такое же как и имя типа (класса)

```
public class Student
{
    private string name;
    private string secondName = "NoName";
    private int course;
    private const string UO = "БГТУ";

    public Student() {
        name = "IR234";
        secondName = "Intel";
        course = 1;
    } //...
}

class Program
{
    static void Main(string[] args)
    {
        Student Olga = new Student();
        Student Serega = new Student();}}}
```



Конструктор без параметров

Свойства конструкторов

- ▶ 3) не наследуются
- ▶ 4) нельзя применять модификаторы `virtual`, `new`, `override`, `sealed` и `abstract`
- ▶ 5) для класса без явно заданных конструкторов компилятор создает конструктор по умолчанию (без параметров)

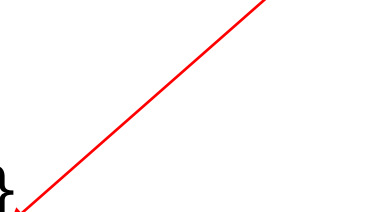
```
public class Student
{
    private string name;
    private string secondName = "NoName";
    private int course;
    private const string UO = "БГТУ";
}

class Program
{
    static void Main(string[] args)
    {
        Student Olga = new Student();
        Student Serega = new Student();}}}
```


Свойства конструкторов

- ▶ 6) для статических классов (запечатанных и абстрактных) компилятор не создает конструктор по умолчанию
- ▶ 7) может определяться несколько конструкторов, сигнатуры и уровни доступа к конструкторам обязательно должны отличаться

```
public class Student
{
    public Student() {}
    public Student(int iCourse) { }
    public Student(string iName){ }
    public Student(string iName, int iCourse) {
}
}
```



Свойства конструкторов

- ▶ 8) можно явно заставлять один конструктор вызывать другой конструктор посредством зарезервированного слова `this`:

```
public class Student
{
    //...
    public Student() {
        name = "IR234";
        secondName = "Intel";
        course = 1;
    }
    public Student(int iCourse):this()
    { }
    public Student(string iName) : this()
    { }
    public Student(string iName, int iCourse) : this()
    { }
}
```

this

- ▶ обеспечивает доступ к текущему экземпляру класса
- ▶ в любой нестатический метод автоматически передается скрытый параметр this

```
public class Student
{
    //...
```

```
    public Student() {
        this.name = "IR234";
        secondName = "Intel";
        course = 0;
    }
    public Student(int course):this()
    {
        course = course;
    }
    //...
```

```
        Student Olga = new Student(20);
        Student Serega = new Student();
        Olga.Info();
        Serega.Info();
```

```
Студент IR234 Intel учится на 0 курсе в БГТУ
Студент IR234 Intel учится на 0 курсе в БГТУ
```

▶ Назначение

- 1) Неоднозначность
- 2) Цепочка конструкторов

```
public class Student
```

```
{
```

```
    //...
```

```
    public Student() {
```

```
        this.name = "IR234";
```

```
        secondName = "Intel";
```

```
        course = 0;
```

```
    }
```

```
    public Student(int course):this()
```

```
    {  
        this.course = course;
```

```
    }
```

```
    //...
```

```
        Student Olga = new Student(20);
```

```
        Student Serega = new Student();
```

```
        Olga.Info();
```

```
        Serega.Info();
```

Неоднозначность

входящий параметр назван так же, как
поле данных данного типа

Студент IR234 Intel учится на 20 курсе в БГТУ

Студент IR234 Intel учится на 0 курсе в БГТУ

Инициализаторы

С помощью инициализатора объектов можно присваивать значения всем доступным полям и свойствам объекта в момент создания без явного вызова конструктора.

```
class Student
{
    public string name ;
    public int age;
}

static void Main(string[] args)
{
    // используем инициализаторы
    Student someStud = new Student
        { name = "Kate", age = 100};
}
```

Деструкторы

- ▶ вызываться непосредственно перед окончательным уничтожением объекта системой "сборки мусора", чтобы гарантировать четкое окончание срока действия объекта.

~имя_класса () { // код деструктора }

нельзя узнать, когда именно следует вызывать деструктор
Если программа завершится до того, как произойдет "сборка мусора", деструктор может быть вообще не вызван

```
~Student()  
{  
    Console.WriteLine("Объект уничтожен");  
}
```

Свойства класса

- ▶ Свойства – специальные методы класса, служат для организации доступа к полям класса.
- ▶ Как правило, свойство связано с закрытым полем класса и определяет методы его получения и установки (предоставляет инкапсуляцию).
- ▶ Синтаксис свойства:

```
[атрибуты]  [спецификаторы]  тип  имясвойства  
{  
    [get код_доступа]  
    [set код_доступа]  
}
```

не void



аксессоры



```
class StudentBSTU
```

```
{
```

```
    private string name;
```

Закрытое поле

Имя - произвольное и не обязательно должно совпадать.

```
    public string Name
```

```
{
```

Свойство

Способ
получения
свойства

```
        get
```

```
{
```

```
            return name;
```

```
}
```

Способ
установки
свойства

```
        set
```

```
{
```

```
            name = value;
```

```
}
```

```
}
```

```
}
```

«умные» поля, то есть полями с дополнительной логикой

представляет передаваемое значение


```
StudentBSTU Марина = new StudentBSTU();
```

Устанавливаем свойство –
срабатывает set

```
Марина.Name = "Марина";
```

"Марина" - передаваемое
в свойство value

```
String nameMar = Марина.Name;
```

Получаем значение свойства
срабатывает блок get

► Назначение - свойства позволяют ВЛОЖИТЬ ДОПОЛНИТЕЛЬНУЮ ЛОГИКУ

```
public int Course
{
    set
    {
        if (value < 1 || value > 4)
        {
            Console.WriteLine("Курс задан не верно");
        }
        else
        {
            course = value;
        }
    }
    get { return course; }
}
```



```
StudentBSTU dima = new StudentBSTU();
dima.Course = 6;
```

Ограничения свойств:

- 1) не может быть передано методу в качестве параметра `ref` или `out`.
- 2) не подлежит перегрузке
- 3) не должно изменять состояние базовой переменной при вызове аксессора `get`
- 4) могут быть статическими, экземпляльными, абстрактными и виртуальными
- 5) могут иметь модификатор доступа
- 6) могут определяться в интерфейсах

```
public int Len  
{
```

```
    get { return Length; }  
    private set { Length = value; }  
} // только один модификатор доступен
```

Set мы сможем использовать только в данном классе - в его методах, свойствах, конструкторе

Автоматические свойства

Имеют сокращенное объявление:

тип имя { get; set; }

компилятор автоматически реализует методы для правильного возвращения значения из поля и назначения значения полю

Проблемы:

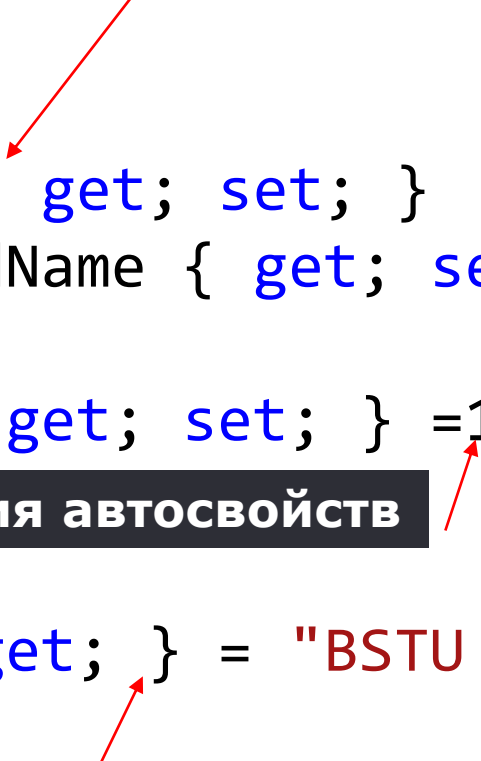
- неявная инициализация
- проблемы при сериализации и десериализации
- во время отладки нельзя установить точку останова

компилятор автоматически генерирует при компиляции поля для свойств

Инициализация значениями по умолчанию

```
public class Student
{
    public string Name { get; set; }
    public string SecondName { get; set; }

    public int Course { get; set; } = 1 ;
    public string UO { get; } = "BSTU ";
}
```



Инициализация автосвойств

для хранения значения этого свойства для него неявно будет создаваться поле с модификатором readonly

C #7

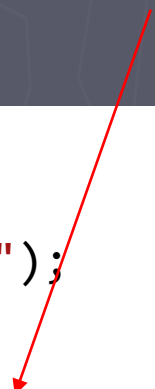
сеттеры, геттеры,
конструкторы и
финализаторы

► expression bodied members

```
public class Person
{
    private StringBuilder names = new StringBuilder("1");

    public Person(string name) => names.Append(name);
                                     // конструктор

    public string Name
    {
        get => names[id];           // геттер
        set => names[id] = value;    // сеттер
    }
}
```



Индексаторы (свойства с параметрами)

- ▶ Позволяют индексировать объекты таким же способом, как массив или коллекцию
- ▶ «умный» индекс для объектов
- ▶ средство, позволяющее разработчику перегружать оператор []

атрибуты спецификаторы тип this

[список_параметров]

get код доступа

set код доступа

```
class SafeArray
{
    public SafeArray(int size)
    {
        a = new int[size];
        length = size;
    }

    public int this[int i]
    {
                // индексатор
        get
        {
            if (i >= 0 && i < length) return a[i];
            else { error = true; return 0; }
        }
        set
        {
            if (i >= 0 && i < length &&
                value >= 0 && value <= 100) a[i] = value;
            else error = true;
        }
    }
    public bool error = false;
    int[] a;
    int length;
    //
}
```



```
//  
static void Main(string[] args)  
{  
    int n = 10;  
    SafeArray sa = new SafeArray(n);  
    for (int i = 0; i < n; ++i)  
    {  
        sa[i] = i * 2;  
    }  
}
```

Индексаторы можно перегружать

Ограничения на индексаторы:

- 1) значение, выдаваемое индексатором, нельзя передавать методу в качестве параметра `ref` или `out`
- 2) индексатор не может быть объявлен как `static`

Многомерные индексаторы

```
class SomeArr
{
    int[,] arr;
    public int rows, cols;
    public int Length;
    // Индексатор
    public int this[int index1, int index2]
    {
        get
        {return arr[index1, index2]; }
        set
        {arr[index1, index2] = value; }
    }
}
```

Особенности хранения ссылочного типа

```
class Program
{
    private static void Main(string[] args)
    {
        Point start = new Point();
    }
}

class Point
{
    public int x;
    public int y;
}
```

// Point - класс, в стек помещается ссылка на адрес в куче

// а в куче располагаются все данные объекта start – работает конструктор по умолчанию

стек

start

куча

start.x
start.y

start = null;

Копирование значений

```
class Program
{
    private static void Main(string[] args)
    {
        Point start = new Point();
        Point end = new Point();
        start = end;
    }
}

class Point
{
    public int x;
    public int y;
}
```

При присвоении данных объекту ссылочного типа он получает не копию объекта, а ссылку на этот объект в куче

Поэтому с изменением end, так же будет меняться start

end
start

end.x
end.y
start.x
start.y

Статические члены класса

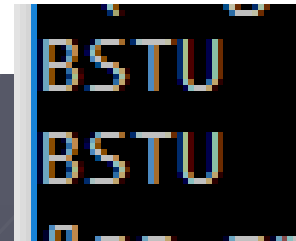
- ▶ переменные и свойства, которые хранят состояние, общее для всех объектов класса, следует определять как статические
- ▶ методы, которые определяют общее для всех объектов поведение, также следует объявлять как статические

```
public class StudentBSTU
{
    private static string uo;
    public static string UO { get; set; } = "BSTU";
    public static void getUo () { Console.WriteLine(UO); }
}
```

- При использовании статических членов необязательно создавать экземпляр класса

```
StudentBSTU.getUo();
```

```
Console.WriteLine(StudentBSTU.UO);
```



- Для статических полей будет создаваться участок в памяти, который будет общим для всех объектов класса.

Свойства статических методов:

- ▶ отсутствует ссылка `this`, поскольку такой метод не выполняется относительно какого-либо объекта
- ▶ в методе `static` допускается непосредственный вызов только других методов типа `static`
- ▶ для метода `static` непосредственно доступными оказываются только другие данные типа `static`, определенные в его классе

Статические конструкторы

или *конструкторы типа*.

Конструктор экземпляра инициализирует данные экземпляра
конструктор класса (типа)— данные класса.

Свойства:

- ▶ закрытые автоматически
- ▶ не имеет параметров
- ▶ нельзя вызвать явным образом (вызываются до создания первого экземпляра объекта и до вызова любого статического метода).

```
class D
{
    private D() { }
    // закрытый конструктор
    static D()
    // статический конструктор
    {
        _a = 200;
    }
    static int _a;
}
```

```
class Point
{
    private static int count;
    public int x;
    public int y = 0; // можно так инициализировать

    static Point()
    {
        count = 0;
        Console.WriteLine("Static constructor");
    }
    public Point()
    {
        count++;
        Console.WriteLine(count);
        Console.WriteLine("Constructor");
    }
}
```

```
Point one = new Point();
```

```
Point two = new Point();
```

```
Point three = new Point();
```

```
Static constructor
1
Constructor
2
Constructor
3
Constructor
```

Статический класс

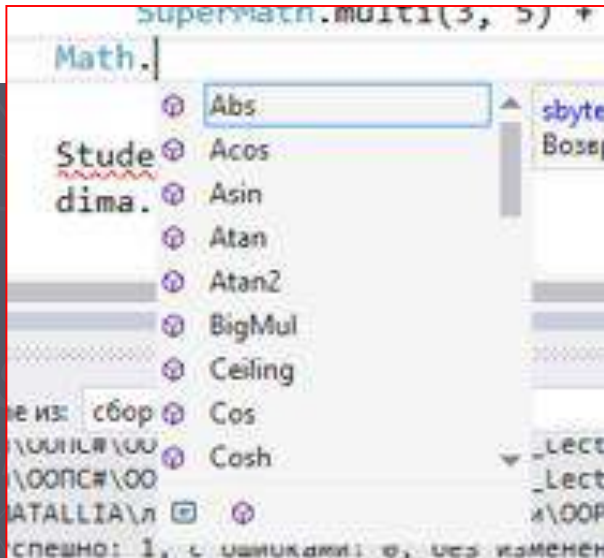
только классы,
но не структуры, CLR всегда
разрешает создавать экземпляры
значимых типов

- ▶ прямой потомок `System.Object`
- ▶ экземпляры такого класса создавать запрещено
- ▶ не должен реализовывать никаких интерфейсов (не вызывать)
- ▶ нельзя использовать в качестве поля, параметра метода или локальной переменной
- ▶ от него запрещено наследовать
- ▶ все элементы такого класса должны явным образом объявляться с модификатором `static`
- ▶ может иметь статический конструктор
- ▶ Компилятор не создает автоматически конструктор по умолчанию

```
static class SuperMath
{
    public static double pi = 3.14;
    public static int multi(int a, int b) => a * b;
    public static int summ(int a, int b) => a + b;
    public static int random() => new Random().Next(100);
}
class Program
{
    static void Main(string[] args)
    {
```

185,14

```
        Console.WriteLine( SuperMath.random()
                            + SuperMath.summ(3, 98)
                            +SuperMath.multi(3, 5) + SuperMath.pi);
    }
```



Назначение:

1) при создании *метода расширения*

2) для хранения совокупности связанных друг с другом статических методов

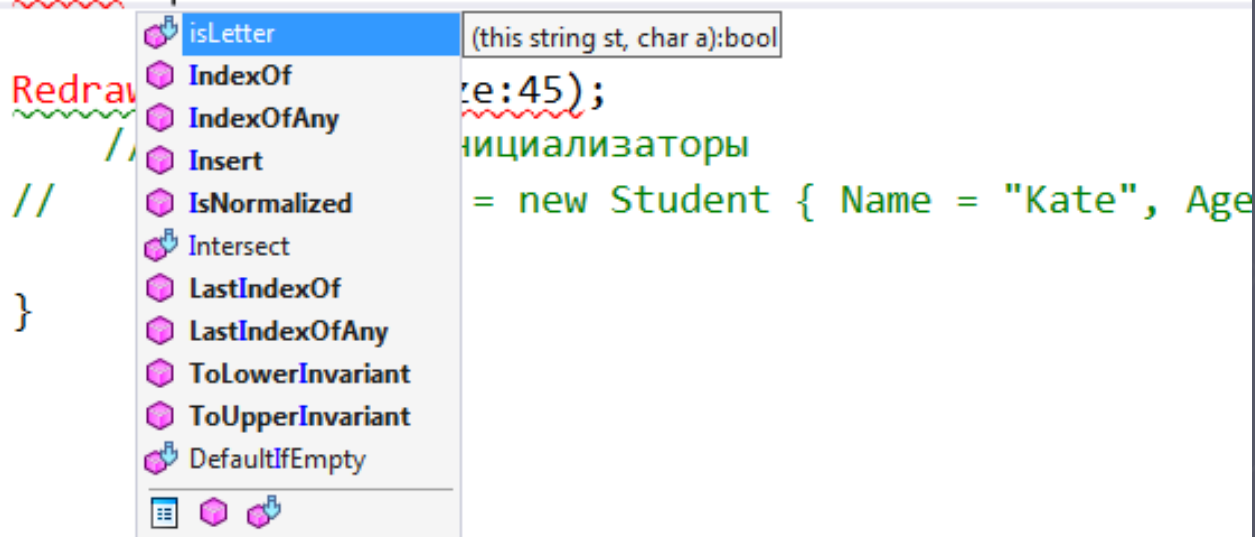
Методы расширения

Методы расширения (extension methods) позволяют добавлять новые методы в уже существующие типы без создания нового производного класса.

```
public static class NewFromAlex
{
    public static bool isLetter(this String st, char a)
    {
        for (Int32 index = 0; index < st.Length; index++)
            if (st[index] == a) return true;
        return false;
    }
}
```

```
String someS = "This test text for extention method";  
someS.isLetter('t');
```

```
someS.i
```



- 1) Проверяется класс и его базовые
- 2) Ищется любой статический класс с методом ####, у которого первый параметр соответствует типу выражения (this)

Правила для методов расширений

- ▶ 1) Методы расширения должны быть объявлены в статическом необобщенном классе (первого уровня)
- ▶ 2) `this` перед первым аргументом и только один
- ▶ 3) использовать аккуратно

Частичные классы

структуры, интерфейсы и методы

Назначение:

Объединение всех частичных файлов класса во время компиляции;
CLR всегда работает с полными определениями типов.

- ▶ Управление версиями
- ▶ Разделение файла или структуры на логические модули
- ▶ Использование шаблонов (авто генерируемый код)

```
internal partial class GoodButton
{ //объявление
    partial void OnClick(int count);
}

internal sealed partial class GoodButton
{ //объявление с реализацией
    partial void OnClick(int count)
    { }
}
```

```
public partial class Student
{
    private string name;
    private string secondName = "NoName";
    private int course;
    private const string UO = "БГТУ";
}
```

```
public partial class Student
{
    public Student() {
        this.name = "IR234";
        secondName = "Intel";
        course = 0;
    }
}
```

Правила использования частичных методов

- ▶ внутри частичного класса или структуры
- ▶ должны всегда иметь возвращаемый тип `void`
- ▶ не могут иметь параметров `out`
- ▶ может иметь параметры `ref`, универсальные параметры, экземплярные или статические, `unsafe`
- ▶ `private` не пишется (закрыт)

АНОНИМНЫЕ ТИПЫ

- ▶ позволяют создать объект с некоторым набором свойств без определения класса (тип в одном контексте или один раз).

не был определен тип имени, автоматически создает имя типа

```
var someType = new {Name = "Anna"};
```

механизм неявной типизации

Используется в
Language Integrated Query, LINQ

компилятор определяет тип каждого выражения

```
<>f__AnonymousType0`1
```

создает закрытые поля
создает открытые свойства только для чтения
создает конструктор → инициализирует
закрытые поля
переопределяет методы Equals, GetHashCode
и ToString

C# 7

Записи - автоматическое создание простых классов

```
class Person (string name, int age);
```

Ссылочный тип Object

- ▶ В CLR каждый объект (и типы значений) прямо или косвенно является производным от `System.Object`

```
// Тип, неявно производный от Object
class Student
{
//...
}

// Тип, явно производный от Object
class Person : System.Object
{
//...
}
```

- ▶ переменная ссылочного типа `object` может ссылаться на объект любого другого типа

```
object helen = new Student();
```

```
object _iValue = 34;
```

```
object _array = new int[4]{2,4, 34,3};
```

Методы System.Object

ToString()

По умолчанию возвращает полное имя типа (`this.GetType().FullName`). Возвращает объект `String`, содержащий состояние объекта в виде строки.

GetHashCode()

Возвращает хеш-код для значения данного объекта

Equals()* и *ReferenceEquals()

Finalize()

Вызывается, когда уборщик мусора определяет, что объект является мусором, но до возвращения занятой объектом памяти в кучу.

GetType()

Возвращает экземпляр объекта, производного от `Type`, который идентифицирует тип объекта

Clone()

Создает новый экземпляр типа и присваивает полям нового объекта значения объекта `this`. Возвращается ссылка на созданный экземпляр

ToString

- ▶ служит для получения строкового представления объекта

```
int year = 2017;  
Console.WriteLine(year.ToString()); // выведет 2017  
Console.WriteLine(3.56.ToString());
```

- ▶ Для классов - выводит полное название класса с указанием пространства имен, в котором определен этот класс.

```
Console.WriteLine(Olga.ToString());
```

```
OOP_Lect.Student
```

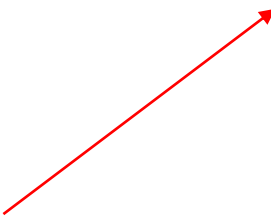
► ToString можем переопределить

```
public partial class Student
{
    public string Name { get; set; }
    public int Course { get; set; } = 1;
    public string UO { get; } = "BSTU";
    public override string ToString()
    {
        return "Type" + base.ToString() + Name + " " + Course + " " + UO;
    }
}
```

```
TypeOOP_Lect.StudentIvan 3 BSTU
```

```
Student ivan = new Student();

ivan.Name = "Ivan";
ivan.Course = 3;
Console.WriteLine(ivan.ToString());
```



Метод Equals

Равенство и тождество объектов

```
public class Object
{
    public virtual Boolean Equals(Object obj)
    {
        // Если обе ссылки указывают на один и тот же объект,
        // значит, эти объекты равны
        if (this == obj) return true;
        // Предполагаем, что объекты не равны
        return false;
    }
}
```

стандартная реализация метода Equals типа Object реализует проверку на тождество

► Корректная реализация

в качестве параметра принимает объект любого типа, который приводим к текущему

```
public class Object
{
    public virtual Boolean Equals(Object obj)
    {
        // Сравниваемый объект не может быть равным null
        if (obj == null) return false;
        // Объекты разных типов не могут быть равны
        if (this.GetType() != obj.GetType()) return false;

        // Если типы объектов совпадают, возвращаем true при условии,
        // что все их поля попарно равны.
        // Так как в System.Object не определены поля,
        // следует считать, что поля равны
        return true;
    }
}
```

Если есть переопределение - этот метод больше не может использоваться для проверки на тождественность

```
public class Object
{
public static Boolean
    ReferenceEquals(Object objA, Object objB)
    {
        return (objA == objB);
    }
}
```

Для проверки на тождественность

System.ValueType (для значимых типов) метод Equals типа Object переопределен и корректно реализован для проверки на равенство (но не тождественность).

Требования к Equals

► Рефлексивность:

- $x.Equals(x) \rightarrow true$

► Симметричность:

- $x.Equals(y)$ и $y.Equals(x) \rightarrow$ результат одинак.

► Транзитивность:

- $x.Equals(y) - true$
- $y.Equals(z) - true$
- $x.Equals(z) \rightarrow true$

► Постоянство:

- не должен измениться если не изменился объект

```
public partial class Student
{
    public string Name { get; set; }
    public int Course { get; set; } = 1;
    public string UO { get; } = "BSTU";

    public override bool Equals(object obj)
    {
        if (obj == null) return false;

        if (obj.GetType() != this.GetType()) return false;

        Student stud = (Student)obj;
        return (this.Name == stud.Name && this.Course == stud.Course)
    }
}
```

```
Student ivan = new Student();
Student oleg = new Student();

Console.WriteLine(oleg.Equals(ivan));
```



True

GetHashCode

Хеш-коды объектов

- ▶ Переопределяется GetHashCode и Equals (парой)

целочисленный (Int32) хеш-код

- ▶ При реализации типов System.Collections.Hashtable, System.Collections.Generic.Dictionary и других коллекций требуется, чтобы два равных объекта имели одинаковые значения хеш-кодов

Требования к GetHashCode

- ▶ Случайное распределение
- ▶ Не использовать GetHashCode для Object или ValueType (низкая производительность алгоритмов хеширования)
- ▶ Использовать экземплярные поля
- ▶ Максимально быстрый
- ▶ Объекты с одинаковым значением должны возвращать одинаковые коды

```
public override int GetHashCode()
{
    // 269 или 47 простые
    int hash = 269;
    hash = string.IsNullOrEmpty(Name) ? 0 : Name.GetHashCode();
    hash = (hash * 47) + Course.GetHashCode();
    return hash;
}
```

```
Console.WriteLine(oleg.GetHashCode());
```

```
TypeOOP_Lect.S  
1579740943
```

метод GetType

- ▶ ПОЗВОЛЯЕТ ПОЛУЧИТЬ ТИП ДАННОГО ОБЪЕКТА

```
Console.WriteLine(ivan.GetType().Name);
```

Возвращает объект
типа Type

получаем тип класса и сравниваем
его с типом объекта

```
if (ivan.GetType() == typeof(Student))  
    Console.WriteLine("Type Student");
```

Finalize()

- ▶ деструктор - вызывается при сборке мусора для очистки ресурсов, занятых ссылочным объектом
- ▶ Реализация из `Objna` игнорируется сборщиком мусора
- ▶ Переопределяется если объект владеет неуправляемыми ресурсами, которые нужно освободить при его уничтожении

Clone()


- ▶ создает копию объекта и возвращает ссылку на эту копию (неглубокое)
- ▶ неглубокое копирование - копируются все типы значений в классе, копируются только ссылки, а не объекты, на которые они указывают
- ▶ не виртуальный, переопределять его реализацию нельзя

Модификаторы параметров методов

для обмена данными между вызывающей и вызываемой функциями предусмотрено четыре типа параметров:

- ▶ По умолчанию- параметры-значения;
- ▶ параметры-ссылки — ref;
- ▶ выходные параметры-ссылки — out;
- ▶ переменное количество — params (один и последний).

```
public int Calculate  
(int a, ref int b, out int c, params int[] d)  
{  
}
```



Назначение:

- ▶ позволить методу менять содержимое его аргументов
- ▶ возвращать более одного значения

► *ref* заставляет C# организовать вместо
ВЫЗОВА ПО ЗНАЧЕНИЮ ВЫЗОВ ПО ССЫЛКЕ

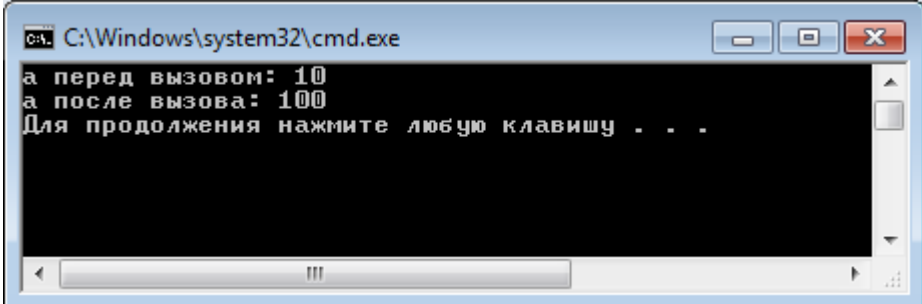
```
class RefTest {  
    public void sqr(ref int i)  
        {i = i * i;}  
}
```

```
public static void Main()  
{ RefTest ob = new RefTest();  
int a = 10;  
  
ob.sqr(ref a);  
}
```

Аргументу, передаваемому методу "в сопровождении"
модификатора **ref**, **должно быть присвоено значение до
вызова метода.**


```
// Использование модификатора ref для передачи  
// значения несылочного типа по ссылке.
```

```
class RefTest  
{  
    // Этот метод изменяет свои аргументы.  
    //Обратите внимание на использование модификатора ref.  
    public void sqr(ref int i)  
        {i = i * i;}  
}  
class RefDemo  
{  
public static void Main()  
    {  
        RefTest ob = new RefTest();  
        int a = 10;  
  
        Console.WriteLine("a перед вызовом: " + a);  
        ob.sqr(ref a);  
        // использование модификатора ref.  
  
        Console.WriteLine("a после вызова: " + a) ;  
    }  
}
```



```
cmd.exe C:\Windows\system32\cmd.exe  
a перед вызовом: 10  
a после вызова: 100  
Для продолжения нажмите любую клавишу . . .
```

out

- Модификатор *out* подобен модификатору *ref* за одним исключением:

его можно использовать для передачи значения из метода

out-параметр "поступает" в метод без начального значения, но метод (до своего завершения) **обязательно** должен присвоить этому параметру значение

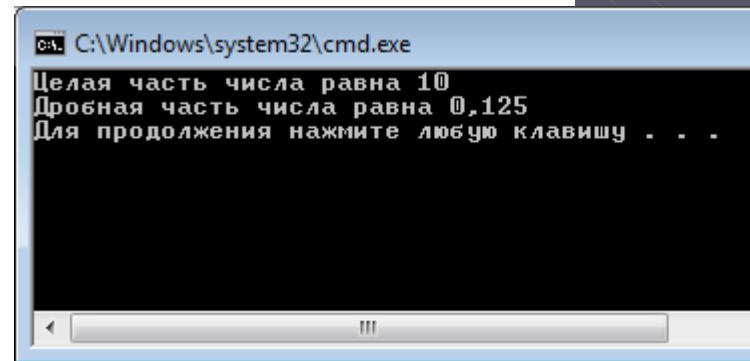
```
class Decompose
{
    //Метод разбивает число с плавающей точкой на
    //целую и дробную части
    public int parts(double n, out double frac)
    {
        int whole;
        whole = (int) n;
        frac = n - whole; // Передаем дробную часть посредством параметра frac.

        return whole; // Возвращаем целую часть числа.
    }
}
```

```
class UseOut
{
    public static void Main()
    {
        Decompose ob = new Decompose();
        int i;
        double f;

        i = ob.parts(10.125, out f);

        Console.WriteLine("Целая часть числа равна " + i);
        Console.WriteLine("Дробная часть числа равна " + f);
    }
}
```



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window displays the output of the program: "Целая часть числа равна 10", "Дробная часть числа равна 0,125", and "Для продолжения нажмите любую клавишу . . .". The text is displayed in a monospaced font on a black background.

```
C:\Windows\system32\cmd.exe
Целая часть числа равна 10
Дробная часть числа равна 0,125
Для продолжения нажмите любую клавишу . . .
```

// Демонстрация использования двух out-параметров.

```
class Num {  
    /*Метод определяет, имеют ли x и y общий делитель.  
    Если да, метод возвращает наименьший и наибольший  
    общие делители в out-параметрах. */  
    public bool isComDenom(int x, int y, out int least, out int greatest)  
    {  
        int i;  
        int max = x < y ? x : y;  
        bool first = true;  
        least = 1;  
        greatest = 1;  
        // Находим наименьший и наибольший общие делители.  
        for (i = 2; i <= max / 2 + 1; i++)  
        {  
            if (((y % i) == 0) & ((x % i) == 0))  
            {  
                if (first)  
                {  
                    least = i;  
                    first = false;  
                }  
                greatest = i;  
                if (least != 1) return true;  
                else return false;  
            }  
        }  
        return false;  
    }  
}
```

```

class DemoOut
{
    public static void Main()
    {
        Num ob = new Num();
        int led, gcd;
        if(ob.isComDenom(231, 105, out led, out gcd))
        {
            Console.WriteLine("Led для чисел 231 и 105 равен " + led) ;
            Console.WriteLine("Gcd для чисел 231 и 105 равен " + gcd) ;
        }

        else
            Console.WriteLine("Для чисел общего делителя нет.");

        if(ob.isComDenom(35, 51, out led, out gcd))
        {
            Console.WriteLine("Led для чисел 35 и 51 равен " + led) ;
            Console.WriteLine("Gcd для чисел 35 и 51 равен " + gcd) ;
        }
        else    Console.WriteLine( "Для чисел общего делителя нет.");}
    }
}

```

C# 7

```
public void ParseXY(Point p)
{
    int x, y; // нужно объявить переменные
    p.ParseXY(out x, out y);
}
```

//C# 7.0

```
public void ParseXY(Point p)
{
    // не нужно объявлять переменные
    p.ParseXY( out int x, out int y);
}
```

Область видимости для таких переменных является внешний блок

```
p.ParseXY( out var x, out var y);
```

params

- позволяет передавать методу переменное количество аргументов одного типа

```
static void MaxArray(ref int _value, params int[] _arr)
{
    if (_arr.Length > 0)
        for (int j = 1; j < _arr.Length; j++)
            if (_arr[j] > _value)
                _value = _arr[j];
}
```

```
static void Check()
{
    int result = -100;
    MaxArray(ref result, 2, 4, 5, 6, 3, 567);
    Console.WriteLine($"Максимум: {result}");
}
```

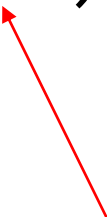


Максимум: 567

Необязательные аргументы

- ▶ позволяет определить используемое по умолчанию значение для параметра метода
- ▶ можно применять в конструкторах, индексаторах

```
static void RedrawButton(int color ,  
                        int type = 2 ,  
                        int size = 4)  
{ }  
static void Main(string[] args)  
{  
    RedrawButton(243);  
}
```



должны указываться
справа от
обязательных

Именованные аргументы

значение аргумента присваивается параметру по его позиции в списке аргументов

позволяет указать имя того параметра, которому присваивается его значение (в конструкторах, индексаторах или делегатах.)

```
static void RedrawButton(int color ,  
                        int type = 2 ,  
                        int size = 4)  
{ }  
static void Main(string[] args)  
{  
    RedrawButton(243, size:45);  
}
```

порядок следования аргументов не имеет значения