Понятие класса, объекта

Введение в ООП

Не ориентированный на объекты подход

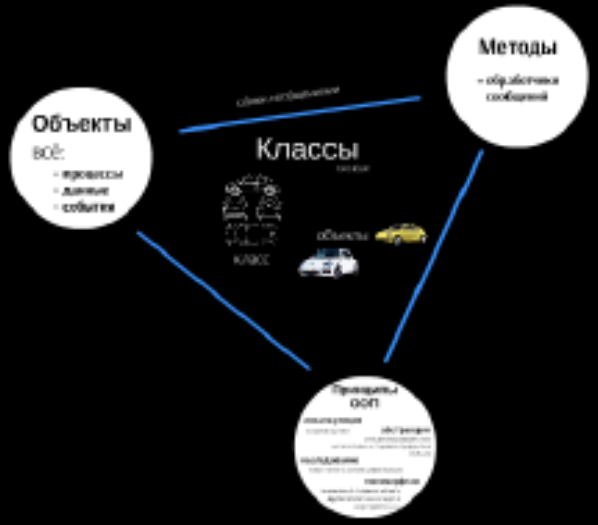
- процедуры
- данные на входе
- вызовы других процедур

```
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

Объектно-ориентированное программирование используется

- структурировать информацию и не допускать путаницы;
- точно определять взаимодействие одних элементов с другими;
- повышать управляемость программы;
- быстрее масштабировать код под различные задачи;
- лучше понимать написанное;
- эффективнее поддерживать готовые программы;
- внедрять изменения без необходимости переписывать весь код.

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

Аналогия

- Понятие «программист» — это класс.
- Конкретный разработчик по имени Иван — это объект, принадлежащий к классу «программист» (экземпляр класса).
- Зарплата, рабочие обязанности, изученные технологии и должность в компании — это свойства, которые есть у всех объектов класса «программист», в том числе у Ивана. У разных объектов свойства различаются: зарплата и обязанности Ивана будут отличаться от таковых у другого разработчика Миши.

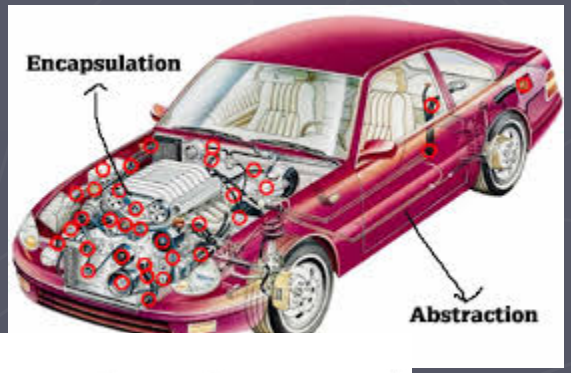
Принципы объектно-ориентированного программирования

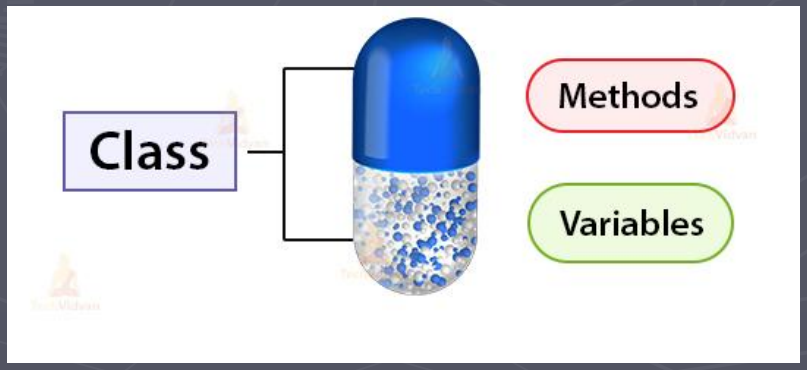
- Инкапсуляция (Encapsulation);
- Наследование (Inheritance);
- Полиморфизм (Polymorphism);
- Абстракция данных(Abstraction).

Инкапсуляция (пакетирование)

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

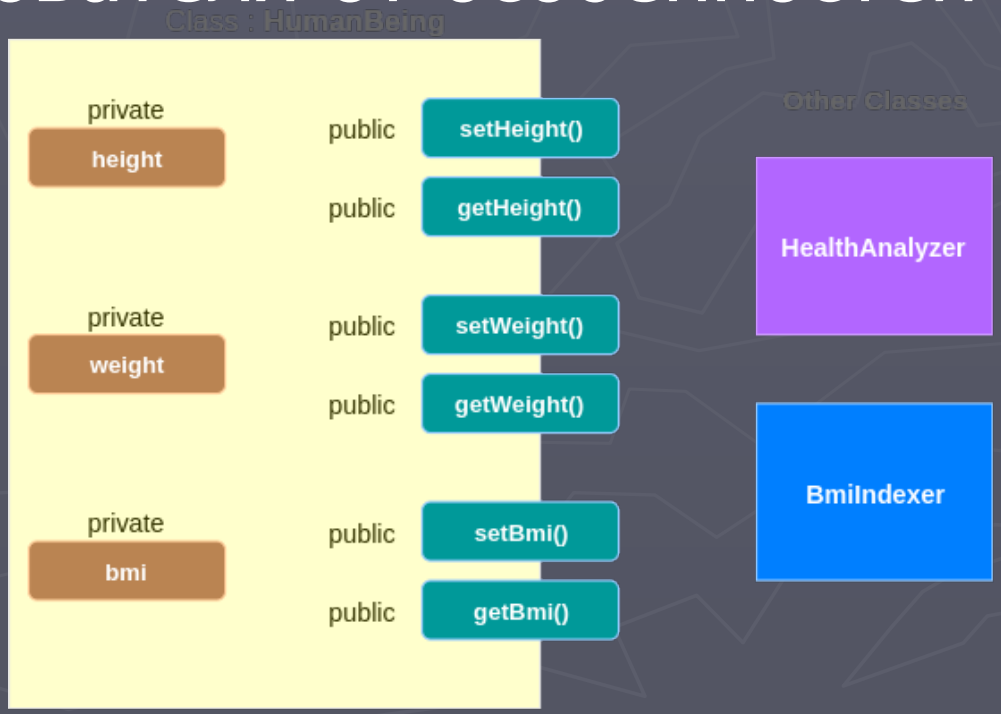
1) Никто не знает что внутри
2) Никто не может менять данные снаружи





Свойства инкапсуляции

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



Абстракция

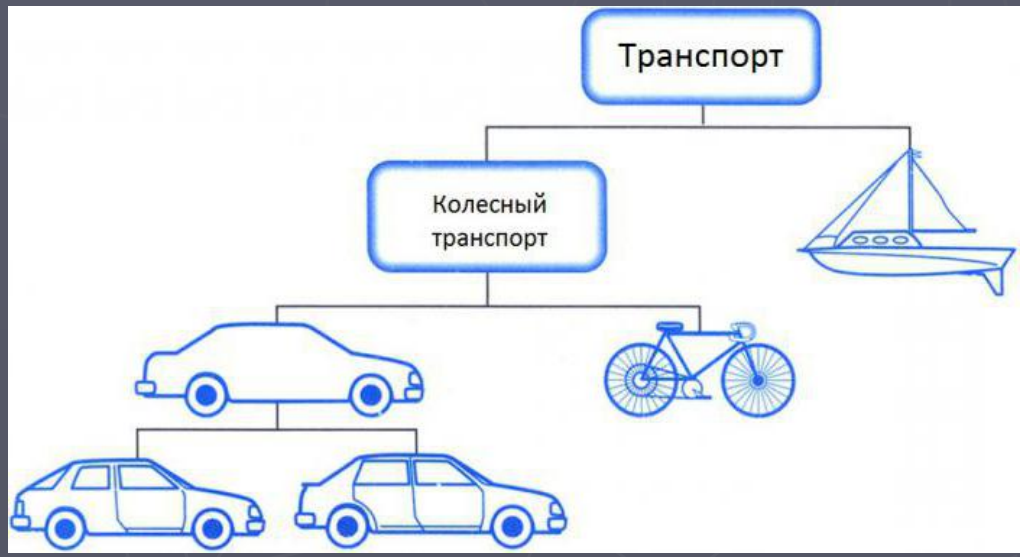
Абстракция подразумевает разделение и независимое рассмотрение **интерфейса** и **реализации**

- способ выделить набор наиболее важных атрибутов и методов и исключить незначимые.

- абстракция - уровень описания/представления модели чего либо

Наследование

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



- *иерархии классов*

У одного «родителя»
может быть несколько дочерних структур.

```
class Phone
{
    public string manufacturer = "Apple";
    public string model = "IPhone XR";

    public void print()
    {
        Console.WriteLine("Мир Apple!");
    }

} class Smartphone : Phone
  {
      public string color = "Red";
  }
```

Базовый класс – класс, от которого наследуется.

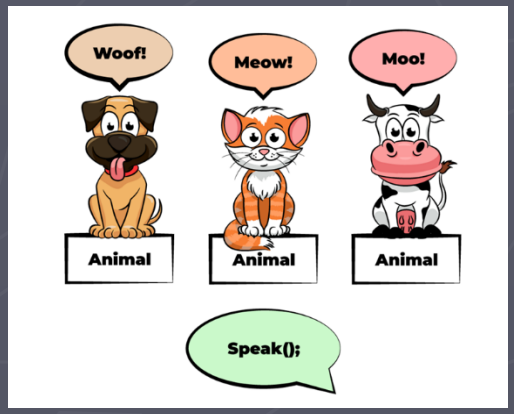
Производный класс – класс, который наследуется от базового класса

реализовывать сложные схемы с четкой иерархией «от общего к частному».

облегчает понимание и масштабирование кода

Полиморфизм

- -поволяет методам классов иметь не одну, а несколько форм, и он необходим, когда у нас есть много классов, связанных друг с другом путем наследования
- - способность программы идентично использовать объекты с одинаковым интерфейсом без информации о конкретном типе этого объекта



Поддержка полиморфизма осуществляется через *виртуальные* функции, механизм перегрузки функций и операторов, а также обобщения

```
class Device
{
    public virtual void enableScreen()
    {
        Console.WriteLine("Screen Enabled!");
    }
}
```

```
class Phone : Device
{
    public override void enableScreen()
    {
        Console.WriteLine("Hello, I am Phone!");
    }
}
```

```
class Tablet : Device
{
    public override void enableScreen()
    {
        Console.WriteLine("Hello, I am Tablet!");
    }
}
```

```
static void Main(string[] args)
{
    Device device = new Device();
    Phone phone = new Phone();
    Tablet tablet = new Tablet();

    device.enableScreen();
    phone.enableScreen();
    tablet.enableScreen();
}
```

Преимущества ООП

Модульность

Объектно-ориентированный подход позволяет сделать **код более структурированным** в нем легко разобраться стороннему человеку.

Благодаря инкапсуляции объектов **уменьшается количество ошибок и ускоряется разработка** с участием большого количества программистов, потому что каждый может работать независимо друг от друга.

Преимущества ООП

Гибкость

ООП-код легко развивать, дополнять и изменять. Это обеспечивает **независимая модульная структура**.

Взаимодействие с объектами, а не логикой упрощает понимание кода. Для модификации **не нужно погружаться в то, как построено ПО.**

Благодаря **полиморфизму** можно быстро адаптировать код под требования задачи, не описывая новые объекты и функции.

Преимущества ООП

Экономия времени

Благодаря абстракции, полиморфизму и наследованию **можно не писать один и тот же код много раз**.

Интерфейсы и классы в ООП могут **легко преобразовываться в подобие библиотек,** которые можно использовать заново в новых проектах.

Безопасность

Программу **сложно сломать,** так как инкапсулированный код недоступен извне.

Недостатки ООП

Сложный старт

нужно сначала изучить теорию и освоить процедурный подход

Снижение производительности

Программы работают несколько медленнее из-за особенностей доступа к данным и большого количества сущностей.

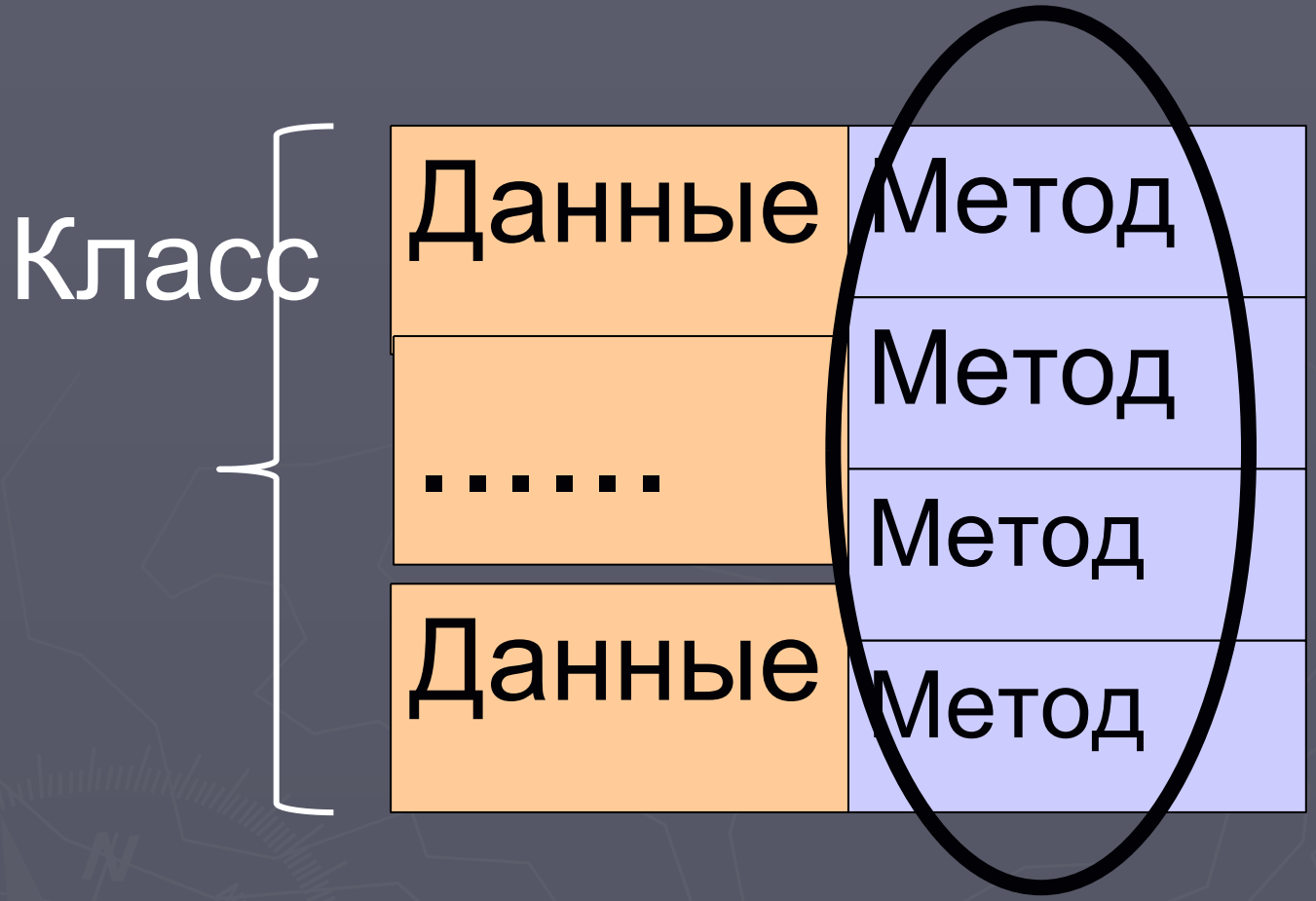
Большой размер программы

ОБЪЕКТ

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

КЛАСС

описание 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).

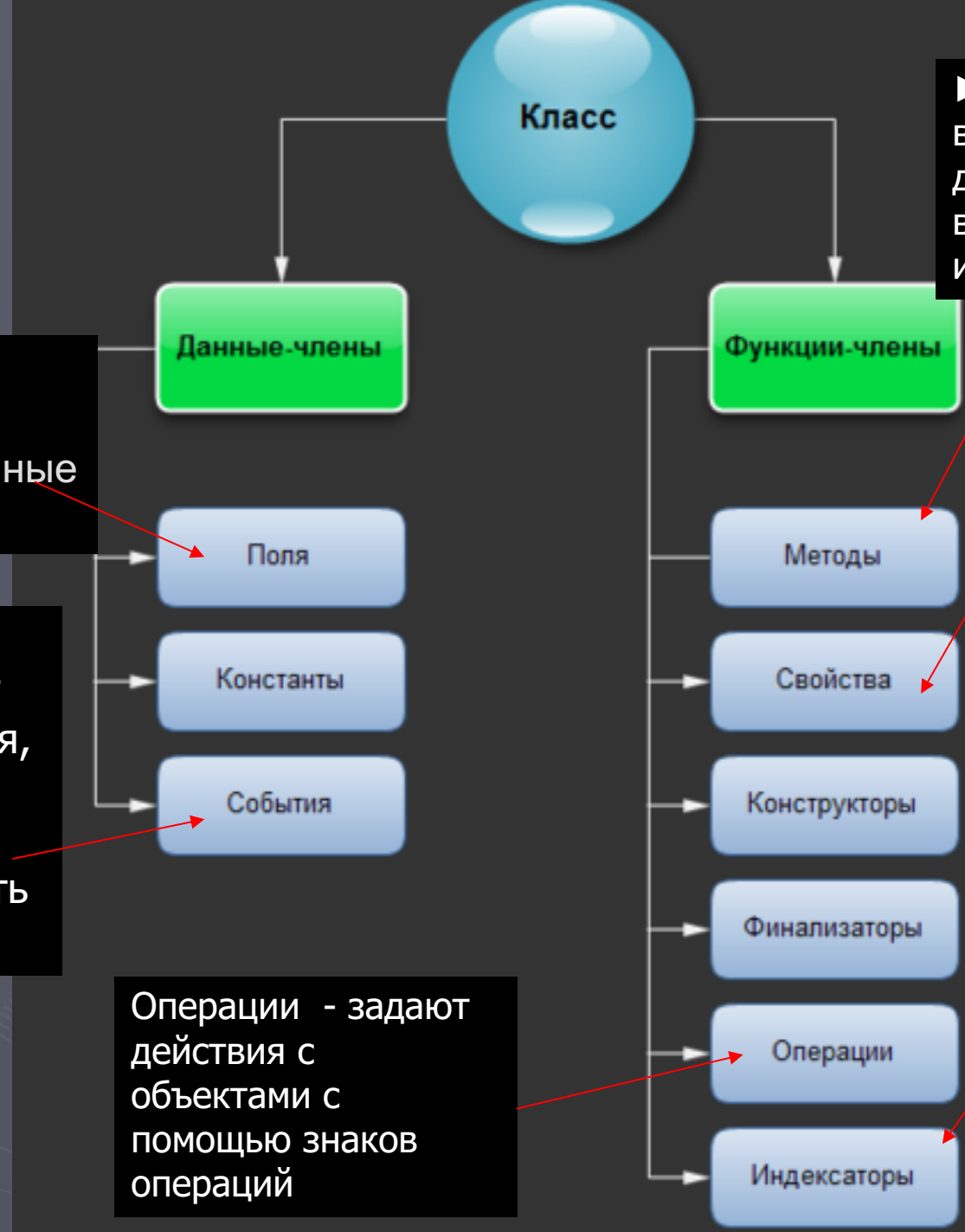
22

Класс в C#

класс представляет **новый тип,** который определяется пользователем.

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

```
class название_класса
{
    // содержимое класса
}
```



любые переменные, ассоциированные с классом

►Методы реализуют вычисления или другие действия, выполняемые классом или экземпляром

Свойства определяют методы записи и чтения

События - определяют уведомления, которые может генерировать класс

Операции - задают действия с объектами с помощью знаков операций

Индексаторы обеспечивают возможность доступа к элементам класса по их порядковому номеру.

```
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

```
Студент  учится на  0
```

Обращение – имя_Объекта.Имя_члена

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
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) определять можем один раз
5)к моменту компиляции они должны быть определены.

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

        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();
        }}
```

Инициализация экземплярного поля (inline)

константа

если мы хотим обратиться к константе вне ее класса, то для обращения необходимо использовались имя класса

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

```
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
             }
}
```

Сравнение констант

	const	readonly
определяются	во время компиляции	во время выполнения программы
инициализировать	при ее определении	либо при его определении, либо в конструкторе класса
static	не могут иметь модификатор static. Неявно уже статические	могут быть как статическими, так и не статическими.

Область видимости (контекст) переменных и констант

- Контекст класса. Переменные, определенные на уровне класса, доступны в любом методе этого класса. Их еще называют глобальными переменными или полями

```
class Person                                    // начало контекста класса
{
    string type = "Person";                     // переменная уровня класса
    public void PrintName()                     // начало контекста метода PrintName
    {
        string name = "Tom";                    // переменная уровня метода

        Console.WriteLine(type);    // в блоке доступна переменная класса
        public void PrintSurname()     // начало контекста метода PrintSurname
    {
        string surname = "Smith";    // переменная уровня метода

        Console.WriteLine(type);       // в методе доступна переменная класса
        Console.WriteLine(surname);// в методе доступна переменная этого же
            метода
    }  // конец конекста метода PrintSurname, переменная surname уничтожается
    }  // конец контекста класса, переменная type уничтожается
```

Область видимости (контекст) переменных и констант

- Контекст метода. Переменные, определенные на уровне метода, являются локальными и доступны только в рамках данного метода. В других методах они недоступны

```
class Person                                    // начало контекста класса
{
    string type = "Person";                     // переменная уровня класса
    public void PrintName()                     // начало контекста метода PrintName
    {
        string name = "Tom";                    // переменная уровня метода

        Console.WriteLine(type);    // в блоке доступна переменная класса
        public void PrintSurname()     // начало контекста метода PrintSurname
    {
        string surname = "Smith";    // переменная уровня метода

        Console.WriteLine(type);       // в методе доступна переменная класса
        Console.WriteLine(surname);// в методе доступна переменная этого же
            метода
    }  // конец конекста метода PrintSurname, переменная surname уничтожается
    }  // конец контекста класса, переменная type уничтожается
```

Область видимости (контекст) переменных и констант

- Контекст блока кода. Переменные, определенные на уровне блока кода, также являются локальными и доступны только в рамках данного блока. Вне своего блока кода они не доступны.

```
class Person                                // начало контекста класса
{
    string type = "Person";                 // переменная уровня класса
    public void PrintName()                 // начало контекста метода PrintName
    {
        string name = "Tom";                // переменная уровня метода

        {                                   // начало контекста блока кода
            string shortName = "Tomas";     // переменная уровня блока кода
            Console.WriteLine(type);        // в блоке доступна переменная класса
 Console.WriteLine(name);   // в блоке доступна переменная окружающего метода
 Console.WriteLine(shortName);// в блоке доступна переменная этого же блока
        }    // конец контекста блока кода, переменная shortName уничтожается

        Console.WriteLine(type);// в методе доступна переменная класса
     Console.WriteLine(name); // в методе доступна переменная этого же метода
//Console.WriteLine(shortName); //так нельзя, переменная c определена в блоке
    кода
//Console.WriteLine(surname);      //так нельзя, переменная surname определена
    в другом методе

    }   // конец контекста метода PrintName, переменная name уничтожаетсяется
```

При работе с переменными надо учитывать, что **локальные переменные**, определенные в методе или в блоке кода, **скрывают переменные уровня класса, если их имена совпадают:**

```
class Person
{
    string name = "Tom";              // переменная уровня класса
    public void PrintName()
    {
        string name = "Tomas";       // переменная уровня метода скрывает переменную уровня класса

        Console.WriteLine(name);     // Tomas
    }
}
```

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 с
борки
    internal class Колесо {  }

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

Доступ к членам типов

- **public** - доступ не ограничен – все члены 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** - **(и**спользуется 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

Модификаторы	Текущий класс	Производный класс из текущей сборки	Производный класс из другой сборки	Непроизводный класс из текущей сборки	Непроизводный класс из другой сборки
private		■	■	■	■
private protected			■	■	■
protected				■	■
internal			■		■
protected internal					■
public					

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

        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();}}
```

►Все поля класса закрытые - спецификатор доступа private

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

►Чаше всего для методов задается спецификатор доступа public

Недоступны из-за уровня защиты

```
public class TestAccess
    {
        int age; // == private int age;
        private int bithday;
                        //  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 addres;
                        // доступно в любом месте программы,
                       //а также для других программ и сборок (dll)
    }
```

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

Перегрузка методов

- один и тот же метод, но с разным набором параметров. методы имеют разную сигнатуру, в которой совпадает только название метода.

- *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**

методы должны отличаться по

- Количеству параметров
- Типу параметров
- Порядку параметров
- Модификаторам параметров

(не включает тип значения, возвращаемого методом, не включает params-параметр)

```
Add(int, int)
Add(int, int, int)
Add(int, int, int, int)
Add(double, double)
```

```
void Increment(ref int val)
{
    val++;
    Console.WriteLine(val);
}

void Increment(int val)
{
    val++;
    Console.WriteLine(val);
}
```

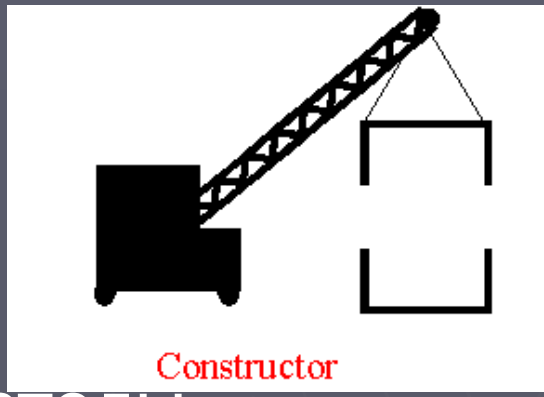
```
// -----------------Перегрузка методов--------
        // Возвращает наибольшее из двух целых:
      class Test{
       public int max( int a, int b ) {return 1;}
        // Возвращает наибольшее из трех целых:
       public int max(int a, int b, int c) { return 2; }
        // Возвращает наибольшее 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));

      }
```

отличие методов по **возвращаемому типу** или **по имени параметров** не является основанием для перегрузки

НЕ ПЕРЕГРУЗКА!

```
int Sum(int x, int y)
{
    return x + y;
}
int Sum(int number1, int number2)
{
    return number1 + number2;
}
void Sum(int x, int y)
{
    Console.WriteLine(x + y);
}
```

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

Память всегда обнуляется до вызова конструктора экземпляра типа. Любые поля, не задаваемые конструктором явно, гарантированно содержат 0 или null.

Свойства конструкторов

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

```
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

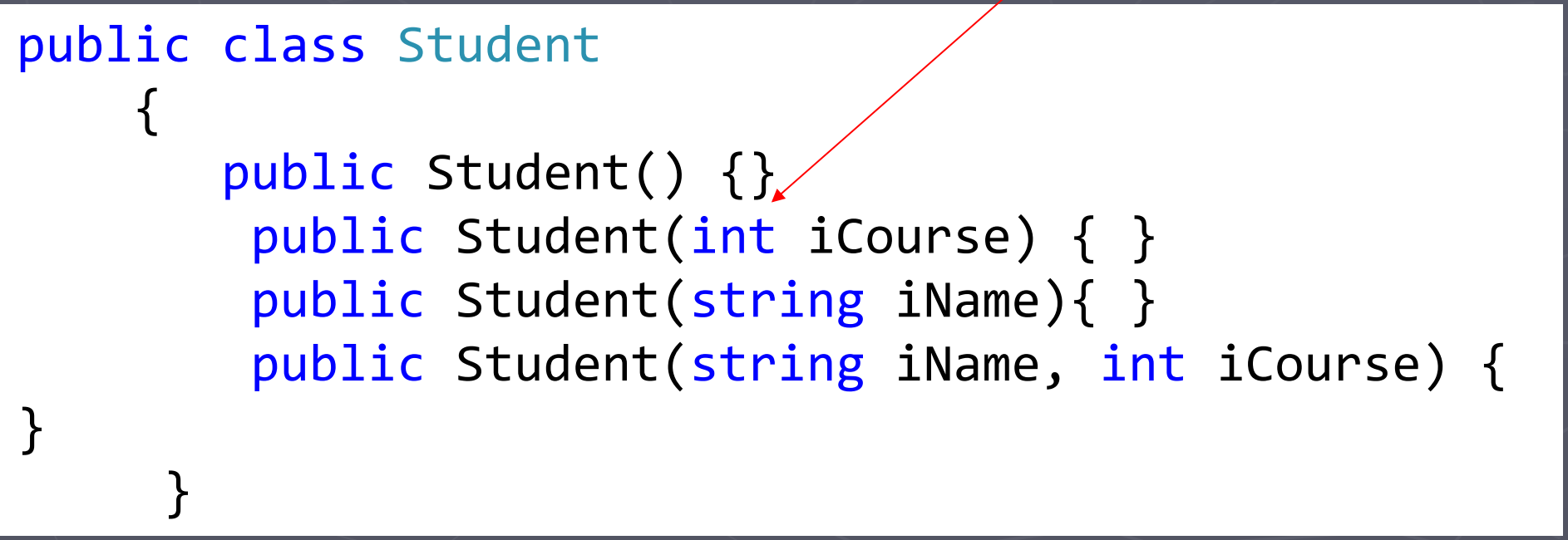
Свойства конструкторов

- 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 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

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

```
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();
```

►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};
    }
```

При использовании инициализаторов следует учитывать следующие моменты:

- С помощью инициализатора мы можем установить значения только **доступных извне класса полей и свойств объекта.**

- Инициализатор в**ыполняется после конструктора**, поэтому если и в конструкторе, и в инициализаторе устанавливаются значения одних и тех же полей и свойств, то значения, устанавливаемые в конструкторе, **заменяются значениями из инициализатора**.

Статические конструкторы

```
public class Child : Person
{
   private static int maximumAge;

   public Child(string lastName, string firstName) : base(lastName, firstName)
   { }

   static Child() => maximumAge = 18;
```

Статические конструкторы

- не должны иметь модификатор доступа и не принимают параметров
- **нельзя** использовать ключевое слово **this** для ссылки на текущий объект класса и можно обращаться только к статическим членам класса
- нельзя вызвать в программе вручную.

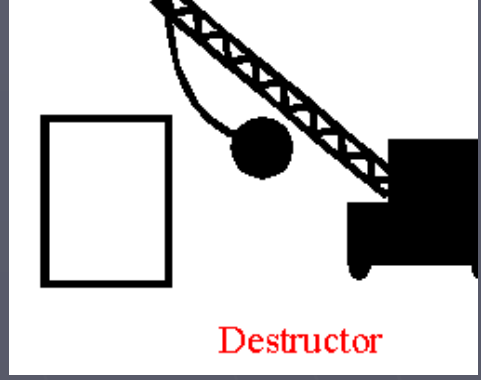
Они выполняются автоматически при самом первом создании объекта данного класса или при первом обращении к его статическим членам

Закрытый конструктор

используется в случаях, если вам необходимо контролировать создание объектов

```
class Axis
{
    public string Name { get; }
    public Vector Direction { get; }
    private Axis(string name, Vector direction) { Name = name; Direction = direction
    static public readonly Axis X = new Axis("X", new Vector(1, 0, 0);
    static public readonly Axis Y = new Axis("Y", new Vector(0, 1, 0);
    static public readonly Axis Z = new Axis("Z", new Vector(0, 0, 1);
}
```

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 вызовется деструктор

Если программа завершиться до того, как произойдет "сборка мусора", деструктор может быть вообще не вызван

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

Свойства деструктора

- Класс может иметь только один деструктор.
- Деструкторы не могут быть унаследованы или перегружены.
- Деструкторы невозможно вызвать. Они запускаются автоматически.
- Деструктор не принимает модификаторы и не имеет параметров.

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:

не void

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

аксессоры

```
class StudentBSTU
    {
        private string name;

        public string Name
        {
            get
            {
                return name;
            }

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

Закрытое поле

Имя - произвольное и не обязательно должно совпадать.

Свойство

Способ получения свойства

Способ установки свойства

«умные» поля, то есть полями с дополнительной логикой

представляет передаваемое значение

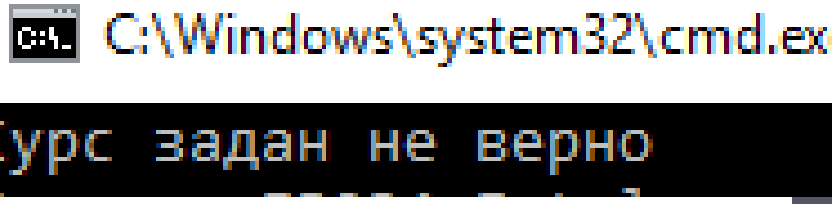
```
StudentBSTU Марина = new StudentBSTU();

Марина.Name = "Марина";

String nameMar = Марина.Name;
```

Устанавливаем свойство –срабатывает set "Марина" - передаваемое в свойство value

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, экземплярными, абстрактными и виртуальными

5) могут иметь модификатор доступа

6) могут определяться в интерфейсах

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

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

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 #6 C#7

- expression bodied members – техника записи метода через лямбда выражения, содержащий один оператор

```
(список_параметров) => выражение
```

```
public Point CreatePoint(int x, int y)
 {
    return new Point(x, y);
 }
```

```
public Point CreatePoint(int x, int y) => new Point(x, y);
```

```
public Point Location
 {
    get
    {
        return _location;
    }
 }
```

```
public Point Location => _location
```

Индексаторы (свойства с параметрами)

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

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

```
class Person
{
    public string Name { get;}
    public Person(string name) => Name=name;
}
class Company
{
    Person[] personal;
    public Company(Person[] people) => personal = people;
    // индексатор
    public Person this[int index]
    {
        get => personal[index];
        set => personal[index] = value;
    }
}
```

Для хранения персонала

Обращаемся к элементам внутри объекта Company

получаем через параметр **value** переданный объект Person и сохраняем его в массив по индексу

```
var microsoft = new Company(new[]
{
    new Person("Tom"), new Person("Bob"), new Person("Sam"), new Person("Alice")
});
// получаем объект из индексатора
Person firstPerson = microsoft[0];
Console.WriteLine(firstPerson.Name);  // Tom
// переустанавливаем объект
microsoft[0] = new Person("Mike");
Console.WriteLine(microsoft[0].Name);  // Mike
```

если индексатору будет передан некорректный индекс, который отсутствует в массиве person, то мы получим исключение,

Индексаторы можно перегружать

также индексаторы должны отличаться по количеству, типу или порядку используемых параметров

Ограничения на индексаторы:

1)значение, выдаваемое индексатором, нельзя передавать методу в качестве параметра ref или out

2) индексатор не может быть объявлен как static

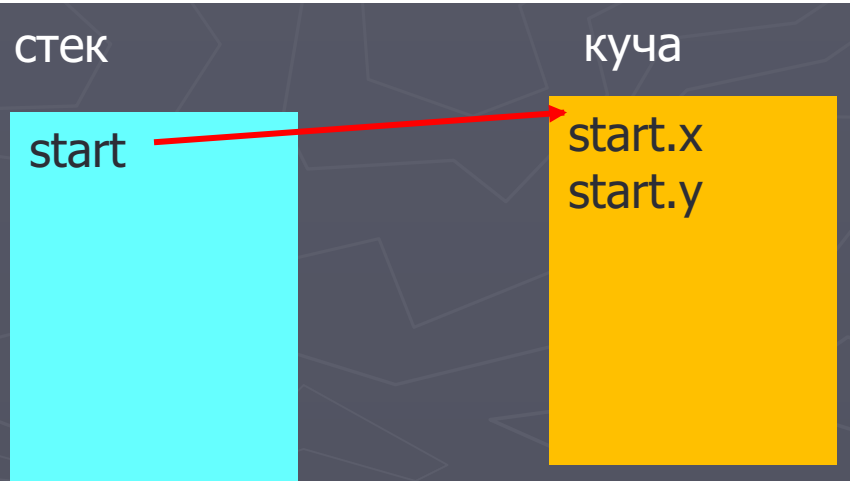
Многомерные индексаторы

```
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
```



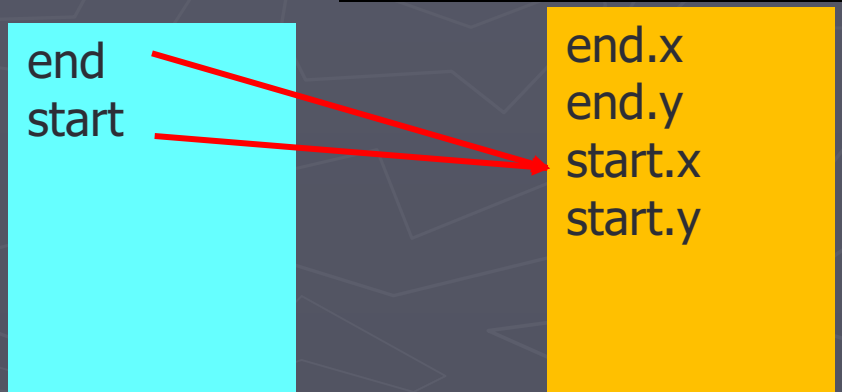

```
start = null;
```

Копирование значений

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



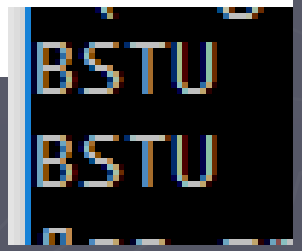
Статические члены класса

- 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

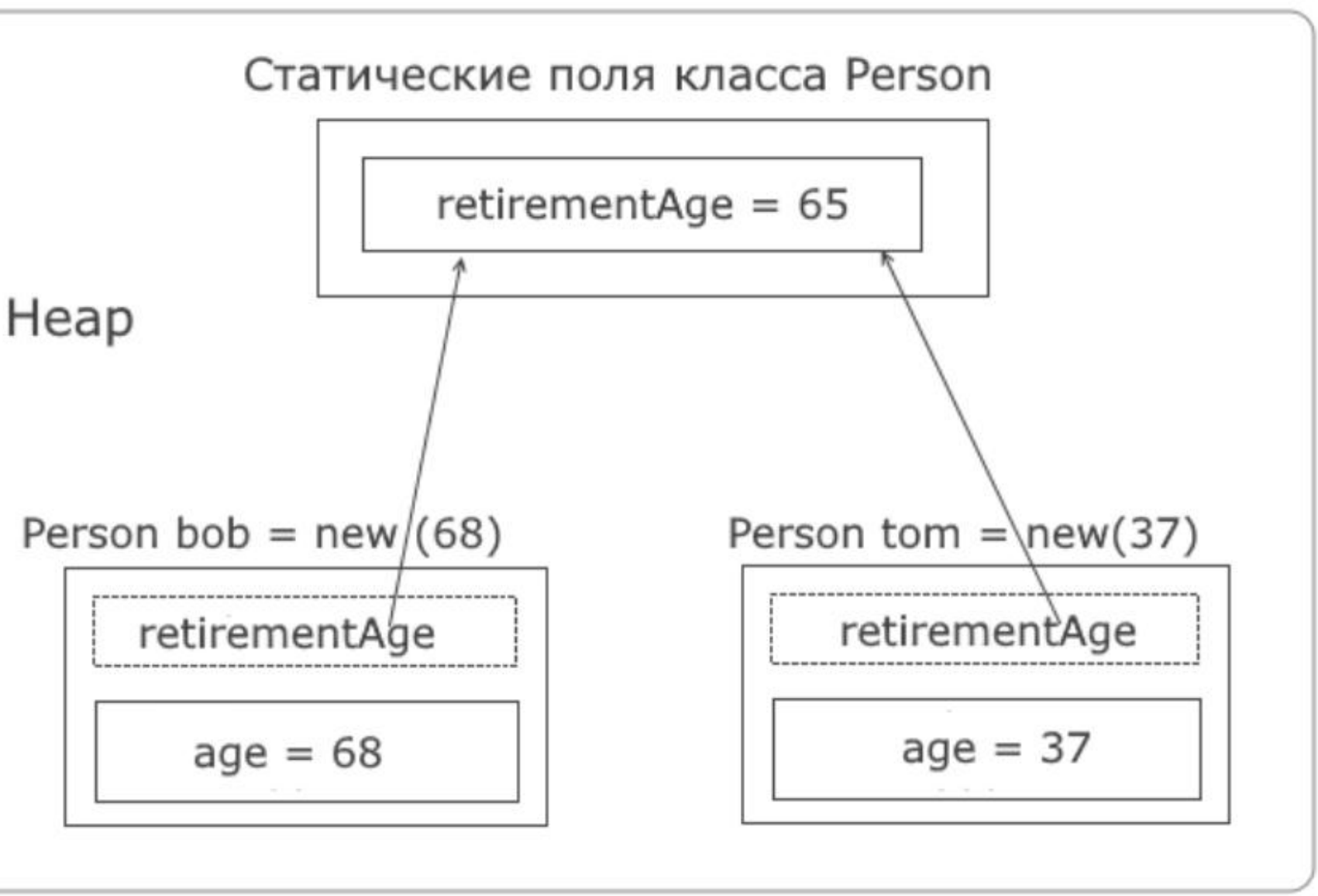
```
public class StudentBSTU
    {
        private static string uo;
        public static string UO { get; set; } = "BSTU";
        public static void getUo ()
         {
             Console.WriteLine(UO);
         }
    }
```

- 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.


Статические поля класса Person
retirementAge = 65
Heap
Person bob = new (68)
retirementAge
age = 68
Person tom = new(37)
retirementAge
age = 37

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 или до вызова любого статического метода).

```
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("Construc
        }
```

```
Point one = new Point();

Point two = new Point();

Point three = new Point();
```

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

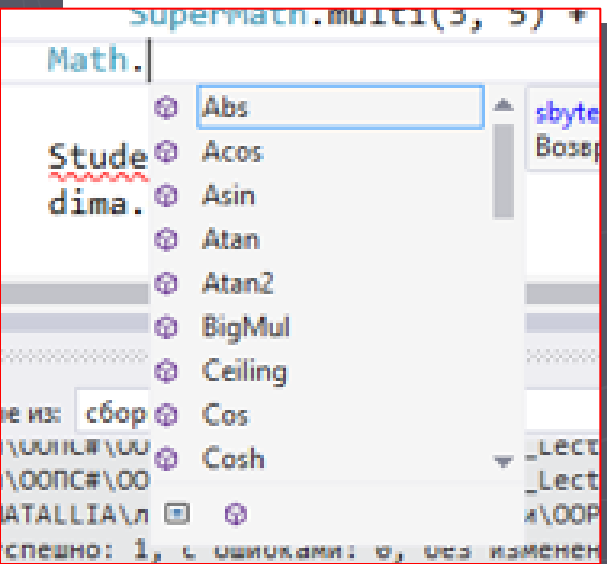
Статический класс

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 вызвать)
- 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)
        {

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

```
185,14
```



Назначение:

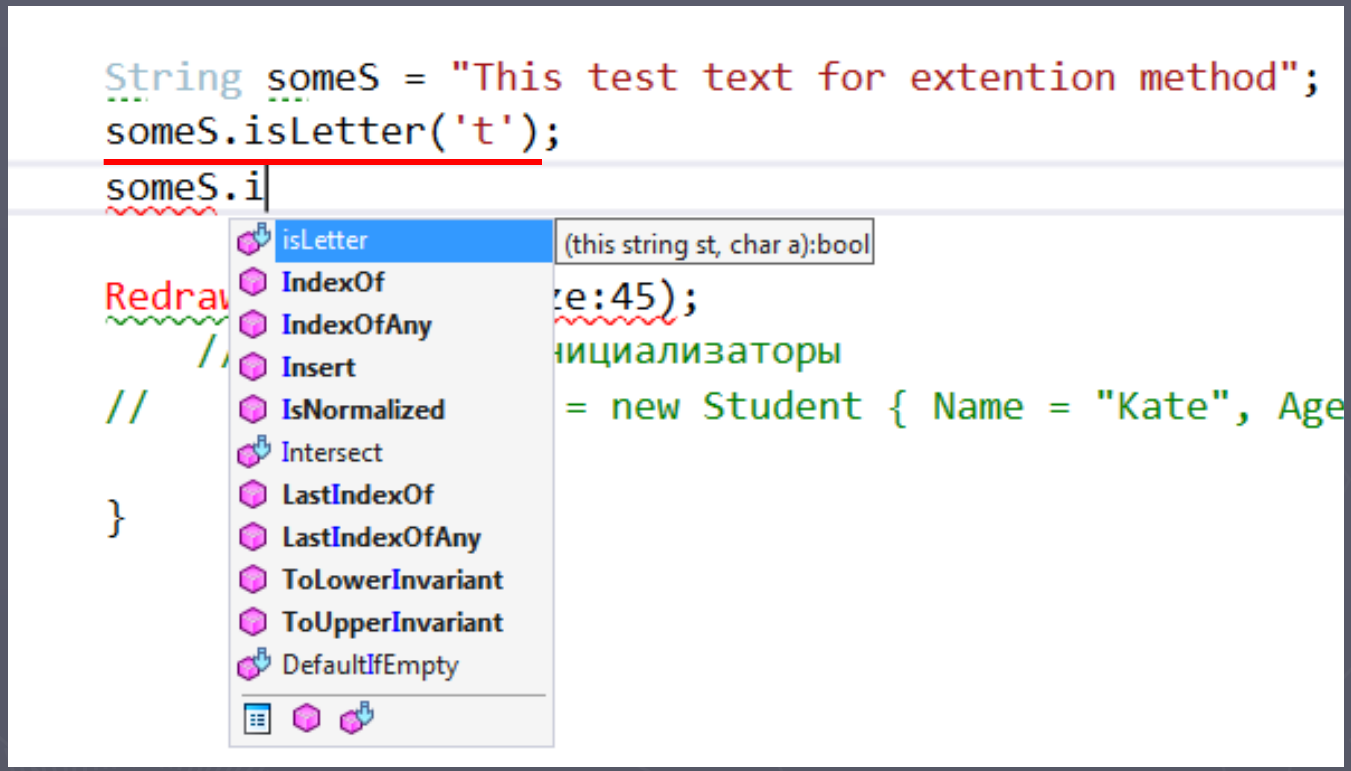
1) при создании *метода расширения*

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 NewStringFunction
{
    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;
       }
 }
```



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) надо помнить, что метод расширения никогда не будет вызван, если он имеет ту же сигнатуру, что и метод, изначально определенный в типе.

Частичные классы

структуры, интерфейсы и методы

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

Назначение:

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

```
internal partial class GoodButton
    {//обьявление
        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)
- нет ни одного из следующих модификаторов: virtual, override, sealed, new или extern.
- В C# 9.0 эти ограничения снимаются, но требуется, чтобы объявления разделяемых методов имели реализацию. Генераторы кода могут предоставить такую реализацию.

Анонимные типы

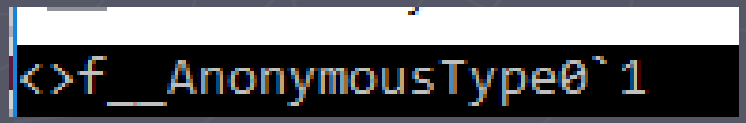
- позволяют создать объект с некоторым набором свойств без определения класса (тип в одном контексте или один раз).

не был определен тип имени, автоматически создает имя типа

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

механизм неявной типизации

компилятор определяет тип каждого выражения



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

Используется в Language Integrated Query, LINQ

Записи (C# 8/9)

это новый ссылочный тип (reference type)

- автоматически поддерживает сравнение экземпляров по значениям их свойств.
- по умолчанию является неизменяемым (immutable).
- позволяет использовать очень компактную форму определения типа.

Записи (C# 8/9)

```
[public | internal] record [имя типа]([список свойств]);
```

Записи - автоматическое создание простых классов

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

```
public record Person (string name, int age);
```

При сравнении записей сравниваются значения а не ссылки.

Записи

```
public record Person {
        public string FirstName { get; set; } = default!;
         public string LastName { get; set; } = default!;
};
```

Запись может наследовать от другой записи. Но запись не может наследовать от класса, а класс не может наследовать от записи.

```
public abstract record Person(string FirstName,
                              string LastName);
public record Teacher(string FirstName,
                      string LastName, int Grade) :
                      Person(FirstName, LastName);
```

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 helena= 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 сборщик 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 = 2021;
Console.WriteLine(year.ToString()); // выведет 2021
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

```
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

Неперегруженный оператор == проверяет для ссылочных типов равенство ссылок. (оба объекта должны указывать на одно значение)

```
object o1 = 5, o2 = 5;
bool eq = (o1 == o2); // false
```

```
object o1 = 5, o2 = o1;
bool eq = (o1 == o2); //true
```

```
object o1 = 5, o2 = 5;
bool eq = o1.Equals(o2); // true
```

метод Equals проверяет равенство значений (оба объекта должны указывать на равное значение).

```
int o1 = 5;
double o2 = 5;
bool eq = (o1 == o2);
Console.WriteLine (eq); //true
bool eq1 = o1.Equals(o2);
Console.WriteLine (eq1); //false
```

сравнивает две ссылки. Если ссылки на объекты идентичны, то возвращает true. Это значит, что данный метод проверяет экземпляры не на равенство, а на тождество.

Для проверки на тождественность

```
 public class Object
        {
public static Boolean
       ReferenceEquals(Object objA, Object objB)
            {
                return (objA == objB);
            }
        }
/////
       int a = 7;
        int b = a;
        bool c=object.ReferenceEquals(a, b);//false
```

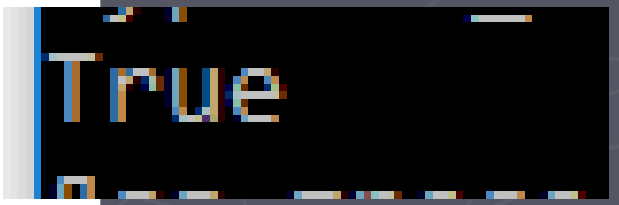
System.ValueType (для значимых типов) метод Equals типа Object переопределен и корректно реализован для
проверки на равенство (но не тождественность).

Требования к Equals

- Рефлексивность:
  - x.Equals(x) // true
- Симметричность:
  - x.Equals(y) и y.Equals(x) // результат одинак.
- Транзитивность:
  - x.Equals(y) - true
  - y.Equals(z) – true
  - x.Equals(z) // true
- Постоянство:
  - Результат 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));
```



GetHashCode
Хеш-коды объектов

- Переопределяется GetHashCode и Equals (парой)

целочисленный (Int32) хеш-код

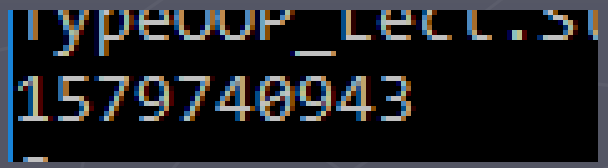
- При реализации типов System.Collections.Hashtable, System.Collections.Generic.Dictionary и других коллекций требуется, чтобы два равных объекта имели одинаковые значения хеш-кодов

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());
```



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 Object 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

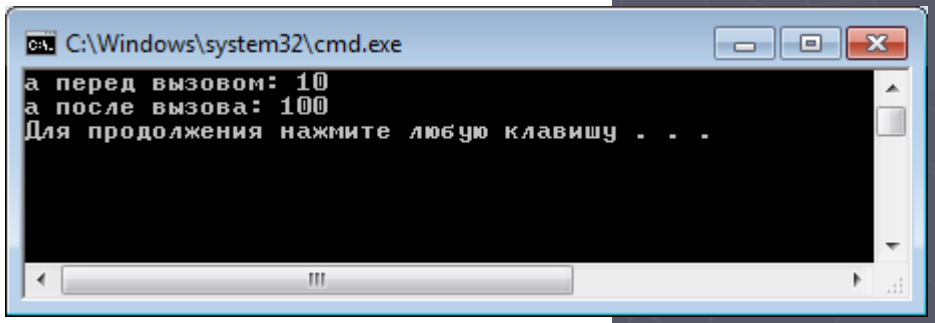
- *ref заставляет C# организовать вместо вызова по значению вызов по ссылке*

```
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 нессылочного 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) ;

          }
  }
```



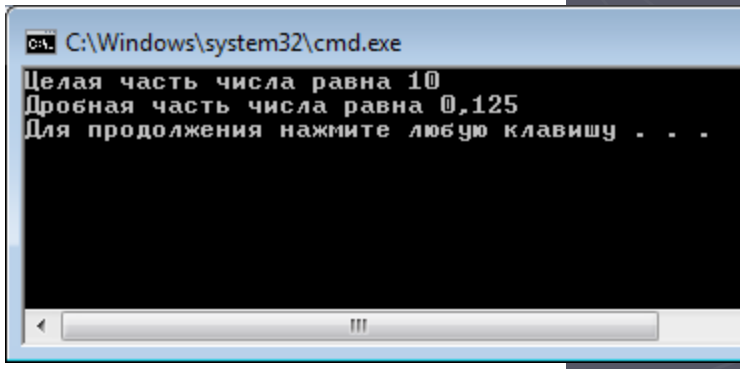
out

- *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);
                              }
     }
```



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}");
        }
```

Тип параметра должен быть одномерным массивом.

- разделенный запятыми список аргументов типа элементов массива;
- массив аргументов указанного типа;
- не передавать аргументы. Если аргументы не отправляются, длина списка params равна нулю.

```
Максимум:567
```

Необязательные аргументы

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

должны указываться справа от обязательных

```
void PrintPerson(string name, int age = 1, string company = "Undefined")
{
    Console.WriteLine($"Name: {name}  Age: {age}  Company: {company}");
}

PrintPerson("Tom", 37, "Microsoft");  // Name: Tom  Age: 37  Company: Microsoft
PrintPerson("Tom", 37);               // Name: Tom  Age: 37  Company: Undefined
PrintPerson("Tom");                   // Name: Tom  Age: 1   Company: Undefined
```

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 имее значения

```
void PrintPerson(string name, int age = 1, string company = "Undefined")
{
    Console.WriteLine($"Name: {name}  Age: {age}  Company: {company}");
}

PrintPerson("Tom", company:"Microsoft", age: 37);  // Name: Tom  Age: 37  Company: Microso
PrintPerson(age:41, name: "Bob");                  // Name: Bob  Age: 41  Company: Undefined
PrintPerson(company:"Google", name:"Sam"); // Name: Sam  Age: 1   Company: Google
```