ООП

Классы и объекты. В чем разница?

Класс – это абстрактный тип данных. С помощью класса описывается некоторая сущность (ее характеристики и возможные действия). Например, класс может описывать студента, автомобиль и т.д. **Описав класс, мы можем создать его экземпляр – объект. Объект – это уже конкретный представитель класса.**

Основные принципы объектно-ориентированного программирования

- **Инкапсуляция – позволяет скрывать внутреннюю реализацию.** В классе могут быть реализованы внутренние вспомогательные методы, поля, к которым доступ для пользователя необходимо запретить, тут и используется инкапсуляция.

- **Наследование – позволяет создавать новый класс на базе другого.** Класс, на базе которого создается новый класс, называется базовым, а базирующийся новый класс – наследником.
- **Полиморфизм – это способность объектов с одним интерфейсом иметь различную реализацию.**

- **Абстракция – позволяет выделять из некоторой сущности только необходимые характеристики и методы, которые в полной мере (для поставленной задачи) описывают объект.** Например, создавая класс для описания студента, мы выделяем только необходимые его характеристики, такие как ФИО, номер зачетной книжки, группа. Здесь нет смысла добавлять поле вес или имя его кота/собаки и т.д.

Общая структура объявления класса

[модификатор доступа] class имя_класса

```
{
  //тело класса
}
```

- ***public*** **– доступ к классу возможен из любого места одной сборки либо из другой сборки, на которую есть ссылка;**
- ***internal*** **– доступ к классу возможен только из сборки, в которой он объявлен.**

режим по умолчанию *internal*.

Класс следует объявлять внутри пространства имен *namespace*, но за пределами другого класса

Пример объявления классов *Student* и *Pupil*:

```
namespace HelloWorld
    {
       class Student //без указания модификатор доступа, класс будет internal
        {
          //тело класса
        }
        public class Pupil
      {
        //тело класса
      }
    }
```

Члены класса

- ***поля*;**
- ***константы*;**
- ***свойства*;**
- ***конструкторы*;**
- ***методы*;**
- ***события*;**
- ***операторы*;**
- ***индексаторы*;**
- ***вложенные типы*.**

Все члены класса, как и сам класс, имеют свой уровень доступа.

- *public* – доступ к члену возможен из любого места одной сборки, либо из другой сборки, на которую есть ссылка;
- *protected* – доступ к члену возможен только внутри класса, либо в классе-наследнике (при наследовании);
- *internal* – доступ к члену возможен только из сборки, в которой он объявлен;
- *private* – доступ к члену возможен только внутри класса;
- *protected internal* - доступ к члену возможен из одной сборки, либо из класса-наследника другой сборки.

по умолчанию режим *private*.

Поля класса

- **Поле – это переменная, объявленная внутри класса**. Как правило, поля объявляются с модификаторами доступа *private* либо *protected*, чтобы запретить прямой доступ к ним. **Для получения доступа к полям следует использовать свойства или методы.**

  Пример объявления полей в классе:

- class Student
  {
    private string firstName;
    private string lastName;
    private int age;
    public string group;
- }

Создание объектов

- **Объявив класс, можно создавать объекты. Делается это при помощи ключевого слова *new* и имени класса:**

- ```
  namespace HelloWorld
  {
    class Student
    {
     …
    }
    class Program
    {
      static void Main(string[] args)
      {
        Student student1 = new Student(); //создание объекта student1 класса Student
        Student student2 = new Student();
      }
    }
  }
  ```
  ```

- **Доступ к членам объекта осуществляется при помощи оператора точка «.» :**

- static void Main(string[] args)
  {
    Student student1 = new Student();
    Student student2 = new Student();

    student1.group = «ИСП-931";
    student2.group = «ИСП-934";

    Console.WriteLine(student1.group); // выводит на экран "Group1"
    Console.Write(student2.group);
    Console.ReadKey();

Такие поля класса *Student*, как *firstName*, *lastName* и *age* указаны с модификатором доступа *private*, поэтому доступ к ним будет запрещен вне класса:

```
static void Main(string[] args)
   {
      Student student1 = new Student();
      student1.firstName= "Nikolay"; //ошибка, нет доступа к полю firstName. Программа не скомпилируется
   }
```

Константы

- **Константы-члены класса ничем не отличаются от простых констант**.

  Константа – это переменная, значений которой нельзя изменить. Константа объявляется с помощью ключевого слова *const*. Пример объявления константы:

- ```
  class Math
  {
    private const double Pi = 3.14;
  }
  ```
  ```

- **Метод – это небольшая подпрограмма, которая выполняет, в идеале, только одну функцию.**

  Методы вместе с полями, являются основными членами класса.

```
class TVSet
{
   private bool switchedOn;

   public void SwitchOn()
   {
     switchedOn = true;
   }
   public void SwitchOff()
   {
     switchedOn = false;
   }
}
class Program
{
   static void Main(string[] args)
   {
     TVSet myTV = new TVSet();
     myTV.SwitchOn(); // включаем телевизор, switchedOn = true;
     myTV.SwitchOff(); // выключаем телевизор, switchedOn = false;
   }
}
```

- **задание**

  Создайте класс Телевизор. В нем есть поле текущий канал. Предусмотрите в нем возможность переключения каналов: следующий канал, предыдущий канал, переход к каналу по номеру. Учтите, что канал не может иметь отрицательный номер.

- **Конструктор – это метод класса, предназначенный для инициализации объекта при его создании.**

Имя всегда совпадает с именем класса.

При объявлении конструктора, не нужно указывать возвращаемый тип, даже ключевое слово *void*.

Конструктор следует объявлять как *public*, иначе объект нельзя будет создать.

В классе всегда существует неявный конструктор по умолчанию, который вызывается при создании объекта с помощью оператора *new*.

Объявление конструктора имеет следующую структуру:

- public [имя_класса] ([аргументы])
  {
    // тело конструктора
  }

```
class Car
    {
      private double mileage;
      private double fuel;

      public Car() //объявление конструктора
      {
        mileage = 0;
        fuel = 0;
      }
    }
    class Program
    {
      static void Main(string[] args)
      {
        Car newCar = new Car(); // создание объекта и вызов
    конструктора
      }
    }
```

Конструктор также может иметь параметры

```
class Car
    {
      private double mileage;
      private double fuel;

      public Car(double mileage, double fuel)
      {
        this.mileage = mileage;
        this.fuel = fuel;
      }
    }
    class Program
    {
      static void Main(string[] args)
      {
        Car newCar = new Car(100, 50); //вызов конструктора с
    параметрами
      }
    }
```

Ключевое слово this

Указатель this - это указатель на объект, для которого был вызван нестатический метод. Ключевое слово *this* обеспечивает доступ к текущему экземпляру класса.

Классический пример использования *this*, это как раз в конструкторах, при одинаковых именах полей класса и аргументов конструктора.

Несколько конструкторов

В классе возможно указывать множество конструкторов, главное чтобы они отличались сигнатурами. Сигнатура, в случае конструкторов, - это набор аргументов

```
class Car
    {
      private double mileage;
      private double fuel;

      public Car()
      {
        mileage = 0;
        fuel = 0;
      }

     public Car(double mileage, double fuel)
     {
       this.mileage = mileage;
       this.fuel = fuel;
     }
    }
```

```
class Program
    {
       static void Main(string[] args)
       {
         Car newCar = new Car(); // создаем автомобиль с параметрами по умолчанию, 0 и 0
         Car newCar2 = new Car(100, 50); // создаем автомобиль с указанными параметрами
       }
    }
```

Если в классе определен один или несколько конструкторов с параметрам нельзя создать объект через неявный конструктор по умолчанию:

```
class Car
    {
      private double mileage;
      private double fuel;

     public Car(double mileage, double fuel)
      {
        this.mileage = mileage;
        this.fuel = fuel;
      }
    }
    class Program
    {
      static void Main(string[] args)
      {
        Car newCar = new Car(100, 50);
        Car newCar2 = new Car(); // ошибка, в классе не определен конструктор
    без параметров
      }
    }
```

- **задание**

  Создайте класс Студент, определите в нем поля: имя, курс, есть ли у него стипендия. Создайте в классе несколько конструкторов, для возможности задания сразу всех указанных параметров или нескольких при создании экземпляров.

Статические члены и модификатор static

Статический метод – это метод, который не имеет доступа к нестатическим полям объекта, и для вызова такого метода не нужно создавать экземпляр (объект) класса, в котором он объявлен.

Статические методы определяют общее для всех объектов поведение, которое не зависит от конкретного объекта. Для обращения к статическим методам применяется имя класса

Простой метод – это метод, который имеет доступ к данным объекта, и его вызов выполняется через объект.

- **Чтобы вызвать простой метод, перед его именем, указывается имя объекта. Для вызова статического метода необходимо указывать имя класса.**

Пример статического метода, который обрезает строку до указанной длины, и добавляет многоточие:

```
class StringHelper
    {
       public static string TrimIt(string s, int max)
       {
         if (s == null)
           return string.Empty;
         if (s.Length <= max)
           return s;

         return s.Substring(0, max) + "...";
       }
    }
```

```
class Program
    {
       static void Main(string[] args)
       {
          string s = "Очень длинная строка, которую
    необходимо обрезать до указанной длины и
    добавить многоточие";
          Console.WriteLine(StringHelper.TrimIt(s, 20));
    //"Очень длинная строка…
       }
    }
```

```
class Person
{
    public int Age { get; set; }
    static int retirementAge = 65;
    public Person(int age) => Age = age;
    public static void CheckRetirementStatus(Person person)
    {
        if (person.Age >= retirementAge)
            Console.WriteLine("Уже на пенсии");
        else
            Console.WriteLine($"Сколько лет осталось до пенсии:
{retirementAge - person.Age}") ;
    }
}

Person bob = new(68);
Person.CheckRetirementStatus(bob);
```

- **Статический метод не имеет доступа к нестатическим полям класса:**
- ```
  class SomeClass
  {
    private int a;
    private static int b;

    public static void SomeMethod()
    {
      a=5; // ошибка
      b=10; // допустимо
    }
  }
  ```
  ```

На применение методов типа static накладывается ряд следующих ограничений:

- **В методе типа static должна отсутствовать ссылка this**, поскольку такой метод не выполняется относительно какого-либо объекта
- **В методе типа static допускается непосредственный вызов только других методов типа static,** но не метода экземпляра из того самого же класса. Дело в том, что методы экземпляра оперируют конкретными объектами, а метод типа static не вызывается для объекта. Следовательно, у такого метода отсутствуют объекты, которыми он мог бы оперировать
- Аналогичные ограничения накладываются на данные типа static. **Для метода типа static непосредственно доступными оказываются только другие данные типа static, определенные в его классе.** Он, в частности, не может оперировать переменной экземпляра своего класса, поскольку у него отсутствуют объекты, которыми он мог бы оперировать

Статические поля

Статические поля хранят состояние всего класса.

```
class Person
{
   int age;
   public static int retirementAge = 65;
   public Person(int age)
   {
      this.age = age;
   }
   public void CheckAge()
   {
      if (age >= retirementAge)
         Console.WriteLine("Уже на пенсии");
      else
         Console.WriteLine($"Сколько лет осталось до пенсии: {retirementAge -
age}");
   }
}
```

- Person bob = new(68);
- bob.CheckAge(); // Уже на пенсии
-
- Person tom = new(37);
- tom.CheckAge(); // Сколько лет осталось до пенсии: 28
-
- // получение статического поля
- Console.WriteLine(Person.retirementAge); // 65
- // изменение статического поля
- Person.retirementAge = 67;

- поле retirementAge относится не к отдельному объекту и хранит значение НЕ отдельного объекта класса Person, а относится ко всему классу Person и хранит общее значение для всего класс
- На уровне памяти для статических полей будет создаваться участок в памяти, который будет общим для всех объектов класса.

Нередко статические поля применяются для хранения счетчиков. Например, пусть у нас есть класс User, и мы хотим иметь счетчик, который позволял бы узнать, сколько объектов User создано:

```
class User
{
    private static int counter = 0;
    public User()
    {
        counter++;
    }

    public static void DisplayCounter()
    {
        Console.WriteLine($"Создано {counter} объектов User");
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        User user1 = new User();
        User user2 = new User();
        User user3 = new User();
        User user4 = new User();
        User user5 = new User();

        User.DisplayCounter(); // 5

        Console.Read();
    }
}
```

- **Статические конструкторы**

- Конструктор можно также объявить как static. **Статический конструктор, как правило, используется для инициализации компонентов, применяемых ко всему классу, а не к отдельному экземпляру объекта этого класса**. Поэтому члены класса инициализируются статическим конструктором до создания каких-либо объектов этого класса.

Статические конструкторы имеют следующие отличительные черты:

- **Статические конструкторы не должны иметь модификатор доступа и не принимают параметров**
- Как и в статических методах, **в статических конструкторах нельзя использовать ключевое слово this для ссылки на текущий объект класса и можно обращаться только к статическим членам класса**
- **Статические конструкторы нельзя вызвать в программе вручную. Они выполняются автоматически при самом первом создании объекта данного класса или при первом обращении к его статическим членам** (если таковые имеются)

```
class User
{
    static User()
    {
        Console.WriteLine("Создан первый пользователь");
    }
}
class Program
{
    static void Main(string[] args)
    {
        User user1 = new User(); // здесь сработает статический конструктор
        User user2 = new User();

        Console.Read();
    }
}
```

Статические классы

Класс можно объявлять как static. **Статический класс обладает двумя основными свойствами. Во-первых, объекты статического класса создавать нельзя. И во-вторых, статический класс должен содержать только статические члены.**

В C# показательным примером статического класса является класс Math, который применяется для различных математических операций.

Свойства

- **Свойство – это член класса, который предоставляет удобный механизм доступа к полю класса (чтение поля и запись). Свойство представляет собой что-то среднее между полем и методом класса. При использовании свойства, мы обращаемся к нему, как к полю класса, но на самом деле компилятор преобразовывает это обращение к вызову соответствующего неявного метода.** Такой метод называется аксессор (*accessor*). Существует два таких метода: *get* (для получения данных) и *set* (для записи).
- **[модификатор доступа] [тип] [имя_свойства]**

```
{
  get
  {
    // тело аксессора для чтения из поля
  }

  set
  {
    // тело аксессора для записи в поле
  }
}
```

Имеется класс *Студент*, и в нем есть закрытое поле курс, которое не может быть ниже единицы и больше пяти. Для управления доступом к этому полю будет использовано свойство *Year*:

```
class Student
{
  private int year; //объявление закрытого поля

  public int Year //объявление свойства
  {
    get // аксессор чтения поля
    {
      return year;
    }
    set // аксессор записи в поле
    {
      if (value < 1)
        year = 1;
      else if (value > 5)
        year = 5;
      else year = value;
    }
  }
}
```

```
class Program
{
   static void Main(string[] args)
   {
     Student st1 = new Student();
     st1.Year = 0; // записываем в поле, используя аксессор set
Console.WriteLine(st1.Year); // читаем поле, используя аксессор get, выведет 1
     Console.ReadKey();
   }
}
```

В теле аксессора *get* может быть более сложная логика доступа, но в итоге **должно возвращаться значение поля, либо другое значение с помощью оператора *return***.

В аксессоре *set* же присутствует неявный параметр *value*, который содержит значение, присваиваемое свойству.

Зачем это нужно?

Если, например, просто сделать
поле *year* открытым и не использовать ни методы, ни свойство для доступа, то можно было бы записать в это поле любое значение, в том числе и некорректное, а так есть **возможность контролировать чтение и запись**.

Для контроля доступа можно было бы здесь использовать простые методы, но для этого бы пришлось реализовать два отдельных метода, с разными именами, и при обращении к ним необходимо использовать скобки, что добавляет лишние неудобства.

```
class Student
{
   private int year;

   public int GetYear()
   {
     return year;
   }
   public void SetYear(int value)
   {
     if (value < 1)
         year = 1;
     else if (value > 5)
         year = 5;
     else year = value;
   }
}
class Program
{
   static void Main(string[] args)
   {
     Student st1 = new Student();
     st1.SetYear(0);
     Console.WriteLine(st1.GetYear());
     Console.ReadKey();
   }
}
```

- **Свойство также может предоставлять доступ только на чтение поля или только на запись. Если, необходимо закрыть доступ на запись, просто не указываем аксессор *set*.** Пример:

- ```
  class Student
  {
    private int year;

    public Student(int y) // конструктор
    {
      year = y;
    }

    public int Year
    {
      get
      {
        return year;
      }
    }
  }
  class Program
  {
    static void Main(string[] args)
    {
      Student st1 = new Student(2);

      Console.WriteLine(st1.Year); // чтение
      st1.Year = 5; // ошибка, свойство только на чтение
      Console.ReadKey();
    }
  }
  ```
  ```

Автоматические свойства

- **Автоматическое свойство – это очень простое свойство, которое, в отличии от обычного свойства, уже определяет место в памяти (создает неявное поле), но при этом не позволяет создавать логику доступа.** Структура объявления Автоматического свойства:

- **[модификатор доступа] [тип] [имя_свойства] { get; set; }**
- **У таких свойств, у их аксессоров отсутствует тело.**

```
class Student
{
   public int Year { get; set; }
}
class Program
{
   static void Main(string[] args)
   {
     Student st1 = new Student();

     st1.Year = 0;
     Console.WriteLine(st1.Year);
     Console.ReadKey();
   }
}
```

- **Автоматически реализуемые свойства есть смысл использовать тогда, когда нет необходимости накладывать какие-либо ограничения на возможные значения неявного поля свойства.**
- **У автоматических свойств остается возможность делать их только на чтение или только на запись. Для этого уже используется модификатор доступа private перед именем аксессора:**

- public int Year { private get; set; } // свойство только на запись
  public int Year { get; private set; } // свойство только на чтение
-

  задание

  Создайте класс *Телевизор*, объявите в нем поле *громкость звука*, для доступа к этому полю реализуйте свойство. Громкость может быть в диапазоне от 0 до 100.

Индексаторы

- Индексаторы позволяют индексировать объекты и обращаться к данным по индексу. Фактически с помощью индексаторов мы можем работать с объектами как с массивами. По форме они напоминают свойства со стандартными блоками get и set, которые возвращают и присваивают значение.

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

- В отличие от свойств индексатор не имеет названия. Вместо его указывается ключевое слово **this**, после которого в квадратных скобках идут параметры. Индексатор должен иметь как минимум один параметр.

Наследование в Си-шарп. Конструктор базового класса

- **В класс-наследник из базового класса переходят поля, свойства, методы и другие члены класса.**

  Объявление нового класса, который будет наследовать другой класс, выглядит так:

- **class [имя_класса] : [имя_базового_класса]**
  {
  // тело класса
  }

```
class Animal
{
   public string Name { get; set; }
}
class Dog : Animal
{
   public void Guard()
   {
     // собака охраняет
   }
}
class Cat : Animal
{
   public void CatchMouse()
   {
     // кошка ловит мышь
   }
}
```

```
class Program
{
  static void Main(string[] args)
   {
     Dog dog1 = new Dog();
     dog1.Name = "Барбос"; // называем пса
     Cat cat1 = new Cat();
     cat1.Name = "Барсик"; // называем кота
     dog1.Guard(); // отправляем пса охранять
     cat1.CatchMouse(); // отправляем кота на
охоту
   }
}
```

Вызов конструктора базового класса в Си-шарп

- **Когда конструктор определен только в наследнике, то при создании объекта сначала вызывается конструктор по умолчанию базового класса, а затем конструктор наследника.**

- **Когда конструкторы объявлены и в базовом классе, и в наследнике – необходимо вызывать их оба. Для вызова конструктора базового класса используется ключевое слово *base*.**

- **[имя_конструктора_класса-наследника] ([аргументы]) : base ([аргументы])**
  {
  // тело конструктора
  }
- В базовый конструктор передаются все необходимые аргументы для создания базовой части объекта

```
class Animal
{
   public string Name { get; set; }

   public Animal(string name)
   {
     Name = name;
   }
}
class Parrot : Animal
{
   public double BeakLength { get; set; } // длина клюва

   public Parrot(string name, double beak) : base(name)
   {
     BeakLength = beak;
   }
}
class Dog : Animal
{
  public Dog(string name) : base (name)
   {
     // здесь может быть логика создания объекта Собака
   }
}
```

```
class Program
{
   static void Main(string[] args)
  {
     Parrot parrot1 = new Parrot("Кеша", 4.2);
     Dog dog1 = new Dog("Барбос");
   }
}
```

Доступ к членам базового класса из класса-наследника

В классе-наследнике можно получить доступ к членам базового класса которые объявлены как *public*, *protected*, *internal* и *protected internal*. Члены базового класса с модификатором доступа *private* также переходят в класс-наследник, но к ним могут иметь доступ только члены базового класса.

Например, свойство, объявленное в базовом классе, которое управляет доступом к закрытому полю, будет работать корректно в классе-наследнике, но отдельно получить доступ к этому полю из класса-наследника нельзя.

задание

Создайте базовый класс *Геометрическая фигура*, предусмотрите в нем общие поля/свойства, например координаты центра фигуры, с помощью конструктора должна быть возможность задать центр. На базе этого класса создайте два новых – *Треугольник* и *Окружность*. В этих классах должны быть свои особые поля, например радиус для окружности. В оба класса добавьте метод *Нарисовать*, в котором могла бы быть специфическая логика рисования фигуры. Создайте объекты треугольник и окружность.

Массив указателей на базовый класс в Си-шарп

```
class Animal
{
  public string Name { get; set; }

  public Animal(string name)
  {
    Name = name;
  }
}
class Dog : Animal
{
  public Dog(string name) : base(name)
  { }

  public void Guard()
  {
    // собака охраняет
  }
}
class Cat : Animal
{
  public Cat(string name) : base(name)
  { }

  public void CatchMouse()
  {
    // кошка ловит мышь
  }
}
```

- ```
  class Program
  {
    static void Main(string[] args)
    {
      List<Animal> animals = new List<Animal>(); // создаем список указателей на базовый класс
      animals.Add(new Dog("Барбос"));
      animals.Add(new Cat("Барсик"));
      animals.Add(new Dog("Полкан"));

      foreach (Animal animal in animals)
      {
        Console.WriteLine(animal.Name);
      }
      Console.ReadLine();
    }
  }
  ```
- Невозможно здесь вызвать методы Guard() или CatchMouse(), но при этом есть доступ к имени животного.

  Обратное невозможно. Нельзя создать массив объектов класса Собака, и записать в него объекты класса Животное.
  ```

Оператор is

Проверяет совместимость объекта с указанным типом (принадлежит ли объект определенному классу). Оператор *is* возвращает истину (true), если объект принадлежит классу. Истина будет также при проверке совместимости объекта класса-наследника и базового класса:

```
static void Main(string[] args)
{
   Dog dog1 = new Dog("Барбос");

   Console.WriteLine(dog1 is Dog); // true
   Console.WriteLine(dog1 is Animal); // true
   Console.WriteLine(dog1 is Cat); // false
   Console.ReadLine();
}
```

Пользуясь оператором *is* и явным преобразованием, теперь можно полноценно использовать массив указателей на базовый класс:

```
class Animal
{
   public string Name { get; set; }

   public Animal(string name)
   {
     Name = name;
   }
}
```

```
class Dog : Animal
{
   public Dog(string name) : base(name)
   { }

   public void Guard()
  {
     Console.WriteLine(Name + " охраняет");
   }
}
class Cat : Animal
{
  public Cat(string name) : base(name)
   { }

   public void CatchMouse()
   {
     Console.WriteLine(Name + " ловит мышь");
   }
}
```

```
class Program
{
   static void Main(string[] args)
  {
     List<Animal> animals = new List<Animal>();
     animals.Add(new Dog("Барбос"));
     animals.Add(new Cat("Барсик"));
     animals.Add(new Dog("Полкан"));

     foreach (Animal animal in animals)
     {
       if (animal is Dog) // проверяем является ли данное животное собакой
          ((Dog)animal).Guard();
       else ((Cat)animal).CatchMouse();
     }
     Console.ReadLine();
   }
}
```

Здесь, использовав явное преобразование, получаем полный доступ к объектам из списка, и можем вызывать методы Guard() и CatchMouse().

Оператор as

В примере выше, вместо явного приведения типов можно было использовать оператор *as*.

***(Dog)animal* эквивалентно выражению *animal as Dog*.**

Разница между оператором *as* и явным приведением в том, что в случае невозможности преобразования, оператор *as* возвращает *null*, тогда как явное приведение выбрасывает исключение.

```
class Program
{
   static void Main(string[] args)
   {
   List<Animal> animals = new List<Animal>();
   animals.Add(new Dog("Барбос"));
   animals.Add(new Cat("Барсик"));
   animals.Add(new Dog("Полкан"));

   foreach (Animal animal in animals)
   {
      if (animal is Dog) // проверяем является ли данное
животное собакой
        (animal as Dog).Guard();
      else (animal as Cat).CatchMouse();
   }
   Console.ReadLine();
   }
}
```

Полиморфизм в Си-шарп

Полиморфизм – это различная реализация однотипных действий.

Классическая фраза, которая коротко объясняет полиморфизм – «Один интерфейс, множество реализаций».

- **Виртуальный метод** – это метод, который МОЖЕТ быть переопределен в классе-наследнике. Такой метод может иметь стандартную реализацию в базовом классе.
- **Абстрактный метод** – это метод, который ДОЛЖЕН быть реализован в классе-наследнике. При этом, абстрактный метод не может иметь своей реализации в базовом классе (тело пустое), в отличии от виртуального.
- **Переопределение метода** – это изменение реализации метода, установленного как виртуальный (в классе наследнике метод будет работать отлично от базового класса).

- Есть класс, в нем объявлен виртуальный или абстрактный метод. От этого класса наследуются еще несколько классов, и в каждом из них по-разному реализуется тот самый виртуальный/абстрактный метод. Получается, объекты этих классов имеют метод с одинаковым именем, но с разной реализацией. В этом и есть полиморфизм.
- Например, есть класс Геометрическая Фигура, и в нем объявлен метод Draw(), который будет рисовать фигуру. От этого класса наследуются классы Треугольник, Прямоугольник, Окружность. В них реализуется метод для рисования (понятно, что реализация рисования каждой фигуры разная). В итоге мы можем создать объекты этих классов, и у всех будет метод Draw(), который будет рисовать соответствующую фигуру.

- Полиморфизм позволяет писать более абстрактные, расширяемые программы, один и тот же код используется для объектов разных классов, улучшается читабельность кода. Полиморфизм позволяет избавить разработчика от написания, чтения и отладки множества if-else/switch-case конструкций.

- **Виртуальный метод объявляется при помощи ключевого слова *virtual*:**

  [модификатор доступа] virtual [тип] [имя метода] ([аргументы])
  {
  // тело метода
  }
- *Статический метод не может быть виртуальным.
- **Объявив виртуальный метод, можно переопределить его в классе наследнике. Для этого используется ключевое слово *override*:**

  [модификатор доступа] override [тип] [имя метода] ([аргументы])
  {
  // новое тело метода
  }

- ```
  class Person
  {
     public string Name { get; set; }
     public int Age { get; set; }

     public Person(string name, int age)
     {
       Name = name;
       Age = age;
     }
     public virtual void ShowInfo() //объявление виртуального метода
     {
       Console.WriteLine("Человек\nИмя: " + Name + "\n" + "Возраст: " + Age + "\n");
     }
  }
  ```
  ```

```
class Student : Person
{
   public string HighSchoolName { get; set; }

   public Student(string name, int age, string hsName)
: base(name, age)
   {
     HighSchoolName = hsName;
   }
   public override void ShowInfo() //
переопределение метода
   {
     Console.WriteLine("Студент\nИмя: " + Name +
"\n" + "Возраст: " + Age +"\n"+ "Название ВУЗа: "
+ HighSchoolName + "\n");
   }
}
```

- ```
  class Pupil : Person
  {
    public string Form { get; set; }

    public Pupil(string name, int age, string form)
  : base(name, age)
    {
      Form = form;
    }
    public override void ShowInfo() // переопределение
  метода
    {
      Console.WriteLine("Ученик(ца)\nИмя: " + Name + "\n"
  + "Возраст: " + Age + "\n" + "Класс: " + Form + "\n");
    }
  }
  ```
  ```

```
class Program
{

   static void Main(string[] args)
   {
     List<Person> persons = new List<Person>();
     persons.Add(new Person("Василий", 32));
     persons.Add(new Student("Андрей", 21, "МГУ"));
     persons.Add(new Pupil("Елена", 12, "7-Б"));

     foreach (Person p in persons)
       p.ShowInfo();

     Console.ReadKey();
   }
}
```

если убрать переопределение, откинув ключевые слова *virtual* и *override,* то в базовом классе и в классе наследнике будут методы с одинаковым именем *ShowInfo*. Программа работать будет, но о каждом объекте, независимо это просто человек или студент/ученик, будет выводиться информация только как о простом человеке (будет вызываться метод *ShowInfo* из базового класса).

Это можно исправить, добавив проверки на тип объекта, и при помощи приведения типов, вызывать нужный метод *ShowInfo*:

```
foreach (Person p in persons)
{
  if (p is Student)
    ((Student)p).ShowInfo();
  else if (p is Pupil)
    ((Pupil)p).ShowInfo();
  else p.ShowInfo();
}
```

Вызов базового метода

Если функционал метода, который переопределяется, в базовом классе мало отличается от функционала, который должен быть определен в классе наследнике то, при переопределении, можно вызвать сначала этот метод из базового класса, а дальше дописать необходимый функционал.

```
public virtual void ShowInfo() // ShowInfo в классе Person
{
   Console.WriteLine("Имя: " + Name);
   Console.WriteLine("Возраст: " + Age);
}

public override void ShowInfo() // ShowInfo в классе Student
{
   base.ShowInfo(); // вызывает базовый метод ShowInfo()
   Console.WriteLine("Название ВУЗа: " + HighSchoolName);
}
```

- задание

Абстрактные классы

Абстрактный класс – это класс объявленный с ключевым словом *abstract*:

```
abstract class [имя_класса]
{
  //тело
}
```

Такой класс имеет следующие особенности:

- нельзя создавать экземпляры (объекты) абстрактного класса;

- абстрактный класс может содержать как абстрактные методы/свойства, так и обычные;

- в классе наследнике должны быть реализованы все абстрактные методы и свойства, объявленные в базовом классе.

Зачем нужны абстрактные классы?

- В самом по себе абстрактном классе, от которого никто не наследуется, смысла нет, так как нельзя создавать его экземпляры. В абстрактном классе обычно реализуется некоторая общая часть нескольких сущностей или другими словами - абстрактная сущность, которая, как объект, не может существовать, и эта часть необходима в классах наследниках

Абстрактные методы

Абстрактный метод – это метод, который не имеет своей реализации в базовом классе, и он ДОЛЖЕН быть реализован в классе-наследнике. Абстрактный метод может быть объявлен только в абстрактном классе.

[модификатор доступа] abstract [тип] [имя метода] ([аргументы]);

- Разница между виртуальным и абстрактным методом

  - Виртуальный метод может иметь свою реализацию в базовом классе, абстрактный – нет (тело пустое);
  - Абстрактный метод должен быть реализован в классе наследнике, виртуальный метод переопределять необязательно.

- Объявление абстрактного метода происходит при помощи ключевого слова *abstract*, и при этом фигурные скобки опускаются, точка с запятой ставится после заголовка метода:

  [модификатор доступа] abstract [тип] [имя метода] ([аргументы]);

- Реализация абстрактного метода в классе наследнике происходит так же, как и переопределение метода – при помощи ключевого слова *override*:

  [модификатор доступа] override [тип] [имя метода] ([аргументы])
  {
    // реализация метода
  }

Абстрактные свойства

- protected [тип] [поле, которым управляет свойство];
  [модификатор доступа] abstract [тип] [имя свойства] { get; set; }
-

  Реализация в классе-наследнике:

  [модификатор доступа] override [тип] [имя свойства]
  {
  get { тело аксессора get }
  set { тело аксессора set }
  }

```
abstract class Animal
{
   public string Name { get; set; }
   public string Type { get; protected set; }

   public abstract void GetInfo(); // объявление абстрактного метода
}
class Parrot : Animal
{
   public Parrot(string name)
   {
     Name = name;
     Type = "Птица";
   }
   public override void GetInfo() // реализация абстрактного метода
   {
     Console.WriteLine("Тип: " + Type + "\n" + "Имя: " + Name + "\n");
   }
}
```

- ```
  class Cat : Animal
  {
    public Cat(string name)
    {
      Name = name;
      Type = "Млекопитающее";
    }
    public override void GetInfo() // реализация абстрактного метода
    {
      Console.WriteLine("Тип: " + Type + "\n" + "Имя: " + Name + "\n");
    }
  }
  class Tuna : Animal
  {
    public Tuna(string name)
    {
      Name = name;
      Type = "Рыба";
    }
    public override void GetInfo() // реализация абстрактного метода
    {
      Console.WriteLine("Тип: " + Type + "\n" + "Имя: " + Name+"\n");
    }
  }
  ```
  ```

- class Program
  {
    static void Main(string[] args)
    {
      List<Animal> animals = new List<Animal>();
      animals.Add(new Parrot("Кеша"));
      animals.Add(new Cat("Пушок"));
      animals.Add(new Tuna("Тёма"));

      foreach (Animal animal in animals)
        animal.GetInfo();

      Console.ReadKey();
    }
  }

- При попытке создать объект абстрактного класса мы получим ошибку "Cannot create an instance of the abstract class or interface 'ConsoleApplication1.Animal'":

  Animal animal = new Animal(); // ошибка

Интерфейсы в Си-шарп. Множественное наследование

Интерфейс представляет собой набор методов (свойств, событий, индексаторов), **реализацию которых должен обеспечить класс, который реализует интерфейс.**

Интерфейс может содержать только сигнатуры (имя и типы параметров) своих членов. Интерфейс не может содержать конструкторы, поля, константы, статические члены. Создавать объекты интерфейса невозможно.

Объявление интерфейса

Интерфейс – это единица уровня класса, он объявляется за пределами класса, при помощи ключевого слова *interface*:

```
interface ISomeInterface
{
  // тело интерфейса
}
```

* Имена интерфейсам принято давать, начиная с префикса «I», чтобы сразу отличать где класс, а где интерфейс.

Внутри интерфейса объявляются сигнатуры его членов, модификаторы доступа указывать не нужно:

```
interface ISomeInterface
{
   string SomeProperty { get; set; } // свойство

   void SomeMethod(int a); // метод
}
```

Реализация интерфейса

- class SomeClass : ISomeInterface // реализация интерфейса ISomeInterface
  {
  // тело класса
  }
-

  Класс, который реализует интерфейс, должен предоставить реализацию всех членов интерфейса:

```
class SomeClass : ISomeInterface
{

  public string SomeProperty
  {
    get
    {
      // тело get аксессора
    }
    set
    {
      // тело set аксессора
    }
  }

  public void SomeMethod(int a)
  {
    // тело метода
  }
}
```

```
interface IGeometrical // объявление интерфейса
{
   void GetPerimeter();
   void GetArea ();
}
```

```
class Rectangle : IGeometrical //реализация интерфейса
{
   public void GetPerimeter()
   {
     Console.WriteLine("(a+b)*2");
   }

   public void GetArea()
   {
     Console.WriteLine("a*b");
   }
}
```

```
class Circle : IGeometrical //реализация интерфейса
{
   public void GetPerimeter()
   {
     Console.WriteLine("2*pi*r");
   }

   public void GetArea()
   {
     Console.WriteLine("pi*r^2");
   }
}
```

```
class Program
{
   static void Main(string[] args)
   {
     List<IGeometrical> figures = new List<IGeometrical>();
     figures.Add(new Rectangle());
     figures.Add(new Circle());
     foreach (IGeometrical f in figures)
     {
       f.GetPerimeter();
       f.GetArea();
     }
     Console.ReadLine();
   }
}
```

Множественное наследование

В C# класс может реализовать сразу несколько интерфейсов.

```
interface IDrawable
{
   void Draw();
}

interface IGeometrical
{
   void GetPerimeter();
   void GetArea ();
}
```

```
class Rectangle : IGeometrical, IDrawable
{
   public void GetPerimeter()
   {
      Console.WriteLine("(a+b)*2");
   }

   public void GetArea()
   {
      Console.WriteLine("a*b");
   }

   public void Draw()
   {
      Console.WriteLine("Rectangle");
   }
}
```

Перегрузка методов в Си-шарп

Это объявление в классе методов с одинаковыми именами при этом с различными параметрами (количество и/или тип).

Отличия только типами возвращаемых значений методами недостаточно для перегрузки

```
public void SomeMethod()
{
  // тело метода
}
public void SomeMethod(int a) // от первого отличается наличием параметра
{
  // тело метода
}
public void SomeMethod(string s) // от второго отличается типом параметра
{
  // тело метода
}
public int SomeMethod(int a, int b) // от предыдущих отличается количеством параметров (плюс изменен тип возврата)
{
  // тело метода
  return 0;
}
```

```
public static void AddAndDisplay(int a, int b)
{
   Console.WriteLine(a + b);
}

public static void AddAndDisplay(char a, char b)
{
   Console.WriteLine(a.ToString() + b.ToString());
}
static void Main(string[] args)
{
   AddAndDisplay(5, 8); // 13
   AddAndDisplay('C', '#'); // "C#"
   Console.ReadKey();
}
```

Перегрузка операторов в Си-шарп

Это реализация своего собственного функционала этого оператора для конкретного класса.

Перегрузка унарного оператора:

- **public static возвращаемый_тип operator оператор (параметры)**
- **{**
  //функционал оператора
- **}**

Модификаторы *public* и *static* являются обязательными, так как перегружаемый оператор будет использоваться для всех объектов данного класса**.**

Возвращаемый тип представляет тот тип, объекты которого мы хотим получить.

На месте [оператор] может стоять любой оператор, который можно перегрузить.

Вместо названия метода идет ключевое слово operator и собственно сам оператор. И далее в скобках перечисляются параметры. Бинарные операторы принимают два параметра, унарные - один параметр. И в любом случае один из параметров должен представлять тот тип - класс или структуру, в котором определяется оператор.

- **Перегрузка бинарного оператора:**

```
public static возвращаемый_тип operator
[оператор]([тип_операнда1]
[операнд1], [тип_операнда2]
[операнд2])
{
  //функционал оператора
}
```

Можно перегружать

Унарные операторы: +, -, !, ++, —, true, false
Бинарные операторы: +, -, *, /, %, &, |, ^, <<, >>, ==, !=, <, >, <=, >=

Нельзя перегружать

[] – функционал этого оператора предоставляют индексаторы
() – функционал этого оператора предоставляют методы преобразования типов
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>= краткие формы оператора присваивания будут автоматически доступны при перегрузке соответствующих операторов (+, -, * …).

```
public class Money
    {
      public decimal Amount { get; set; }
      public string Unit { get; set; }

      public Money(decimal amount, string unit)
      {
        Amount = amount;
        Unit = unit;
      }
public static Money operator +(Money a, Money b) //перегрузка
    оператора «+»
      {
        if (a.Unit != b.Unit)
          throw new InvalidOperationException("Нельзя
    суммировать деньги в разных валютах");

        return new Money(a.Amount + b.Amount, a.Unit);
      }
```

```
Money myMoney = new Money(100, "USD");
Money yourMoney = new Money(100, "RUR");
Money hisMoney = new Money(50, "USD");
Money sum = myMoney + hisMoney; // 150 USD
sum = yourMoney + hisMoney; // исключение
    - разные валюты
```

```
public static Money operator ++(Money a) //
   перегрузка «++»
     {
       a.Amount++;
       return a;
     }
     public static Money operator --(Money a) //
   перегрузка «--»
     {
       a.Amount--;
       return a;
     }
```

```
Money myMoney = new Money(100, "USD");
        myMoney++; // 101 USD
```

Также **существует возможность перегрузки самого операторного метода**. Это означает что в классе может быть несколько перегрузок одного оператора при условии что входные параметры будут отличаться типом данных

```
public static Money operator +(Money a, Money b)
    {
      if (a.Unit != b.Unit)
        throw new InvalidOperationException("Нельзя
  суммировать деньги в разных валютах");

      return new Money(a.Amount + b.Amount, a.Unit);
    }
    public static string operator +(string text, Money a)
    {
      return text + a.Amount + " " + a.Unit;
    }
```

```
Money myMoney = new Money(100, "USD");
        Console.WriteLine("У меня сейчас " +
    myMoney); // "У меня сейчас 100 USD"
```

- реализовать второй вариант сложения денег – чтобы можно было суммировать деньги в разных валютах. Для этого создайте отдельный класс, который будет предоставлять механизм конвертации денег по заданному кур

Класс System.Object и его методы

- Все остальные классы в .NET, даже те, которые мы сами создаем, а также базовые типы, такие как **System.Int32**, являются неявно производными от класса Object. Поэтому все типы и классы могут реализовать те методы, которые определены в классе System.Object. Рассмотрим эти методы.

ToString

- Метод ToString служит для получения строкового представления данного объекта. Для базовых типов просто будет выводиться их строковое значение:

- int i = 5;
- Console.WriteLine(i.ToString()); // выведет число 5
-
- double d = 3.5;
- Console.WriteLine(d.ToString()); // выведет число 3,5

Для классов же этот метод выводит полное название класса с указанием пространства имен, в котором определен этот класс. Мы можем переопределить данный метод. Посмотрим на примере:

```
class Clock
    {
        public int Hours { get; set; }
        public int Minutes { get; set; }
        public int Seconds { get; set; }
        public override string ToString()
        {
            return $"{Hours}:{Minutes}:{Seconds}";
        }
    }
    class Person
    {
        public string Name { get; set; }
    }
```

- Person person = new Person { Name = "Tom" };
- Console.WriteLine(person.ToString()); // выведет название класса Person
-
- Clock clock = new Clock { Hours = 15, Minutes = 34, Seconds = 53 };
- Console.WriteLine(clock.ToString()); // выведет 15:34:53

```
public override string ToString()
  {
     if (String.IsNullOrEmpty(Name))
        return base.ToString();
     return Name;
  }
```

- если имя - свойство Name не имеет значения, оно представляет пустую строку, то возвращается базовая реализация - название класса. Если же имя установлено, то возвращается значение свойства Name. Для проверки строки на пустоту применяется метод String.IsNullOrEmpty().

- метод Console.WriteLine() по умолчанию выводит именно строковое представление объекта. Поэтому, если нам надо вывести строковое представление объекта на консоль, то при передаче объекта в метод Console.WriteLine необязательно использовать метод ToString() - он вызывается неявно:
- Clock clock = new Clock { Hours = 15, Minutes = 34, Seconds = 53 };
- Console.WriteLine(clock); // выведет 15:34:53

Метод GetHashCode

- Метод **GetHashCode** позволяет возвратить некоторое числовое значение, которое будет соответствовать данному объекту или его хэш-код. По данному числу, например, можно сравнивать объекты. Можно определять самые разные алгоритмы генерации подобного числа или взять реализацию базового типа:

```
class Person
{
    public string Name { get; set; }

    public override int GetHashCode()
    {
        return Name.GetHashCode();
    }
}
```

- В данном случае метод GetHashCode возвращает хеш-код для значения свойства Name. То есть два объекта Person, которые имеют одно и то же имя, будут возвращать один и тот же хеш-код. Однако в реальности алгоритм может быть самым различным

Получение типа объекта и метод GetType

- Метод GetType позволяет получить тип данного объекта:
- Person person = new Person { Name = "Tom" };
- Console.WriteLine(person.GetType()); // Person

- Этот метод возвращает объект **Type**, то есть тип объекта.
- С помощью ключевого слова **typeof** мы получаем тип класса и сравниваем его с типом объекта. И если этот объект представляет тип Person, то выполняем определенные действия.

- object person = new Person { Name = "Tom" };
- if (person.GetType() == typeof(Person))
- Console.WriteLine("Это реально класс Person")

- Причем поскольку класс Object является базовым типом для всех классов, то мы можем переменной типа object присвоить объект любого типа. Однако для этой переменной метод GetType все равно вернет тот тип, на объект которого ссылается переменная. То есть в данном случае объект типа Person.
- В отличие от методов ToString, Equals, GetHashCode метод GetType не переопределяется.

Метод Equals

- Метод Equals позволяет сравнить два объекта на равенство:
- class Person
- {
- public string Name { get; set; }
- public override bool Equals(object obj)
- {
- if (obj.GetType() != this.GetType()) return false;
-
- Person person = (Person)obj;
- return (this.Name == person.Name);
- }
- }

- Метод Equals принимает в качестве параметра объект любого типа, который мы затем приводим к текущему, если они являются объектами одного класса. Затем сравниваем по именам. Если имена равны, возвращаем true, что будет говорить, что объекты равны. Однако при необходимости реализацию метода можно сделать более сложной, например, сравнивать по нескольким свойствам при их наличии.
- Применение метода

- Person person1 = new Person { Name = "Tom" };
- Person person2 = new Person { Name = "Bob" };
- Person person3 = new Person { Name = "Tom" };
- bool p1Ep2 = person1.Equals(person2); // false
- bool p1Ep3 = person1.Equals(person3); // true

Регулярные выражения в Си-шарп. Класс Regex

Это некий шаблон, составленный из символов и спецсимволов, который позволяет находить подстроки соответствующие этому шаблону в других строках.

Что можно делать, используя регулярные выражения

- заменять в строке все одинаковые слова другим словом, или удалять такие слова;
- выделять из строки необходимую часть. Например, из любой ссылки (http://mycsharp.ru/post/33/2013_10_19_virtualnye_metody_v_si-sharp_pereopredelenie_metodov.html) выделять только доменную часть (mycsharp.ru);
- проверять соответствует ли строка заданному шаблону. Например, проверять, правильно ли введен email, телефон т.д.;
- проверять, содержит ли строка заданную подстроку;
- извлекать из строки все вхождения подстрок, соответствующие шаблону регулярного выражения. Например, получить все даты из строки.

необходимо подключить в начале программы пространство имен ***using System.Text.RegularExpressions;***

В Си-шарп работу с регулярными выражениями предоставляет класс Regex. **Создание регулярного выражения имеет следующий вид:**

Regex myReg = new Regex([шаблон]);

Здесь [шаблон] – это строка содержащая символы и спецсимволы.

У *Regex* также есть и второй конструктор, который принимает дополнительный параметр – опции поиска.

Методы класса Regex

IsMatch **– проверяет содержит ли строка хотя бы одну подстроку соответствующую шаблону регулярного выражения.**

Match **– возвращает первую подстроку, соответствующую шаблону, в виде объекта класса Match. Класс Match предоставляет различную информацию о подстроке – длину, индекс, само значение и другое.**

```
string data1 = "Петр, Андрей, Николай";
string data2 = "Петр, Андрей, Александр";
Regex myReg = new Regex("Николай"); // создание регулярного выражения
 Console.WriteLine(myReg.IsMatch(data1)); // True
 Console.WriteLine(myReg.IsMatch(data2)); // False
```

IsMatch проверят, содержит ли заданная строка (data1, data2) подстроку соответствующую шаблону.

```
string data1 = "Петр, Андрей, Николай";
   Regex myReg = new Regex("Николай");
   Match match = myReg.Match(data1);
   Console.WriteLine(match.Value); //
"Николай"
   Console.WriteLine(match.Index); // 14
```

***Matches* – возвращает все подстроки соответствующие шаблону в виде коллекции типа *MatchCollection*. Каждый элемент этой коллекции типа *Match*.**

```
 string data1 = "Петр, Николай, Андрей, Николай";
 Regex myReg = new Regex("Николай");
 MatchCollection matches = myReg.Matches(data1);
 Console.WriteLine(matches.Count); // 2
 foreach (Match m in matches)
     Console.WriteLine(m.Value); //вывод всех подстрок "Николай"
```

***Replace* – возвращает строку, в которой заменены все подстроки, соответствующие шаблону, новой строкой:**

```
string data1 = "Петр, Николай, Андрей, Николай";
   Regex myReg = new Regex("Николай");
   data1 = myReg.Replace(data1, "Максим");
   Console.WriteLine(data1); //"Петр, Максим, Андрей, Максим"
```

***Split* - возвращает массив строк, полученный в результате разделения входящей строки в местах соответствия шаблону регулярного выражения:**

```
string data1 = "Петр,Николай,Андрей,
Николай";
   Regex myReg = new Regex(",");
   string[] names = myReg.Split(data1); //
массив имен
```

Специальные символы

Обозначение	Описание	Шаблон	Соответствие
[группа_символов]	Любой из перечисленных в скобках символов. Используя тире можно указать диапазон символов, например, [a-f] - то же самое, что [abcdef]	[abc]	«a» в «and»
[^группа_символов]	Любой символ, кроме перечисленных в скобках	[^abc]	«n», «d» в «and»
\d	Цифра. Эквивалентно [0-9]	\d	«1» в «data1»
\D	Любой символ, кроме цифр. Эквивалентно [^0-9]	\D	«y» в «2014y»
\w	Цифра, буква (латинский алфавит) или знак подчеркивания. Эквивалентно [0-9a-zA-Z_]	\w	«1», «5», «c» в «1.5c»
\W	Любой символ, кроме цифр, букв (латинский алфавит) и знака подчеркивания. Эквивалентно [^0-9a-zA-Z_]	\W	«.» в «1.5c»
\s	Пробельный символ (пробел, табуляция, перевод строки и т. п.)	\s	« » в «c sharp»
\S	Любой символ, кроме пробельных	\S	«c» «s» «h» «a» «r» «p» в «c sharp»
.	Любой символ, кроме перевода строки. Для поиска любого символа, включая перевод строки, можно использовать конструкцию [\s\S]	c.harp	«csharp» в «mycsharp»

Символы повторения

Обозначение	Описание	Шаблон	Соответствие
*	Соответствует предыдущему элементу ноль или более раз	\d*.	«a», «1b», «23c » в «a1b23c»
+	Соответствует предыдущему элементу один или более раз	\d+.	«1b», «23c » в «a1b23c»
?	Соответствует предыдущему элементу ноль или один раз	\d?\D	«a», «1b», «3c» в «a1b23c»
{n}	Соответствует предыдущему элементу, который повторяется ровно n раз	\d{2}	«43», «54», «82» в «2,43,546,82»
{n,}	Соответствует предыдущему элементу, который повторяется минимум n раз	\d{2,}	«43», «546», «82» в «2,43,546,82»
{n,m}	Соответствует предыдущему элементу, который повторяется минимум n раз и максимум m	\d{2,}	«43», «546», «821» в «2,43,546,8212»

Символы выбора

Обозначение	Описание	Шаблон	Соответствие
|	Работает как логическое «ИЛИ» - соответствует первому и/или второму шаблону	one|two	«one», «two» в «one two three»
(группа_символов)	Группирует набор символов в единое целое для которого дальше могут использоваться + * ? и т.д. Каждой такой группе назначается порядковый номер слева направо начиная с 1. По этому номеру можно ссылаться на группу \номер_группы	(one)\1	«oneone» в «oneone onetwoone»
(?:группа_символов)	Та же группировка только без назначения номера группы	(?:one){2}	«oneone» в «oneone onetwoone»

Другие символы

Обозначение	Описание	Шаблон	Соответствие
\t	Символ табуляции	\t	
\v	Символ вертикальной табуляции	\v	
\r	Символ возврата каретки	\r	
\n	Символ перевода строки	\n	
\f	Символ перевода страницы	\f	
\	Символ, который позволяет экранировать специальные символы, чтобы те воспринимались буквально. Например, чтобы было соответствие символу звёздочки, шаблон будет выглядеть так *	\d\.\d	«1.1», «1.2» в «1.1 1.2»

Символы повторения

Символы привязки

Символы выбора

Другие символы

```
string s = "Бык тупогуб, тупогубенький бычок, у быка губа бела была тупа";
Regex regex = new Regex(@"\w*губ\w*");
```

Так как выражение \w* соответствует любой последовательности алфавитно-цифровых символов любой длины, то данное выражение найдет все слова, содержащие корень "губ".

Нахождение телефонного номера в формате 111-111-1111:

```
string s = "456-435-2318";
Regex regex = new Regex(@"\d{3}-\d{3}-\d{4}");
```

Если мы точно знаем, сколько определенных символов должно быть, то мы можем явным образом указать их количество в фигурных скобках: \d{3} - то есть в данном случае три цифры.

Regex regex = new Regex("[a-v]{5}"); - данное выражение будет соответствовать любому сочетанию пяти символов, в котором все символы находятся в диапазоне от a до v

Regex regex = new Regex(@"[2]*-[0-9]{3}-\d{4}");. Это выражение будет соответствовать, например, такому номеру телефона "222-222-2222" (так как первые числа двойки)

С помощью операции | можно задать альтернативные символы:

```
Regex regex = new
Regex(@"[2|3]{3}-[0-9]{3}-\d{4}");
```

То есть первые три цифры могут содержать только двойки или тройки. Такой шаблон будет соответствовать, например, строкам "222-222-2222" и "323-435-2318". А вот строка "235-435-2318" уже не подпадает под шаблон, так как одной из трех первых цифр является цифра 5.

Если надо найти, строки, где содержится точка, звездочка или какой-то другой специальный символ, то в этом случае надо просто экранировать эти символы слешем:

```
Regex regex = new
Regex(@"[2|3]{3}\.[0-9]{3}\.\d{4}");
// этому выражению будет
соответствовать строка "222.222.2222"
```

Можно не только задать поиск по определенным типам символов - пробелы, цифры, но и задать конкретные символы, которые должны входить в регулярное выражение.

```
string s = "456-435-2318";

Regex regex = new
Regex("[0-9]{3}-[0-9]{3}-[0-9]{4}");
```

В квадратных скобках задается диапазон символов, которые должны в данном месте встречаться.

Проверка на соответствие строки формату

```
Regex myReg = new
Regex(@"[A-Za-z]+[\.A-Za-z0-9_-]*[A-Za-z0-9]+@[A-Za-z]+\.[A-Za
-z]+");
Console.WriteLine(myReg.IsMatch("email@email.com")); // True
Console.WriteLine(myReg.IsMatch("email@email")); // False
 Console.WriteLine(myReg.IsMatch("@email.com")); // False
```

```
string s = "Мама  мыла  раму. ";
string pattern = @"\s+";
string target = " ";
Regex regex = new Regex(pattern);
string result = regex.Replace(s, target);
```

Данная версия метода Replace принимает два параметра: строку с текстом, где надо выполнить замену, и сама строка замены. Так как в качестве шаблона выбрано выражение "\s+ (то есть наличие одного и более пробелов), метод Replace проходит по всему тексту и заменяет несколько подряд идущих пробелов ординарными.

Параметры поиска

Второй конструктор *Regex* принимает в качестве второго аргумента значение перечисления *RegexOptions*. В этом перечисление есть следующие значения:

IgnoreCase – игнорирование регистра при поиске. Находит соответствия независимо прописными или строчными буквами в строке написано слово;
RightToLeft – поиск будет выполнен справа налево, а не слева направо;
Multiline – многострочный режим поиска. Меняет работу спецсимволов «^» и «$» так, что они соответствуют началу и концу каждой строки, а не только началу и концу целой строки;
Singleline – однострочный режим поиска;
CultureInvariant - игнорирование национальных установок строки;
ExplicitCapture – обеспечивается поиск только буквальных соответствий;
Compiled – регулярное выражение компилируется в сборку, что делает более быстрым его исполнение но увеличивает время запуска;
IgnorePatternWhitespace – игнорирует в шаблоне все неэкранированные пробелы. С этим параметром шаблон «a b» будет аналогичным шаблону «ab»;

```
string data = "nikolay, sergey, oleg";
Regex myRegIgnoreCase = new Regex(@"Sergey",
RegexOptions.IgnoreCase);
Regex myReg = new Regex(@"Sergey");
 Console.WriteLine(myRegIgnoreCase.IsMatch(data)); //
True
  Console.WriteLine(myReg.IsMatch(data)); // False
```

Если необходимо установить несколько параметров, тогда они разделяются оператором поразрядного «ИЛИ» - «|»

```
Regex myReg = new Regex(@"Sergey",
RegexOptions.IgnoreCase |
RegexOptions.IgnorePatternWhitespace);
```

Задания

- **Удалить числа (цифры) из текста**
- Получить все знаки препинания из строки, кроме пробела.
- Найти количество «не» в предложении
- Убрать все не из строки