ПРОГРАММИРОВАНИЕ НА C#
часть 2

Класс и объект

Язык C# является объектно-ориентированным языком программирования.
Любую программу (проект) на языке C# можно представить в виде одного или нескольких объектов, которые могут взаимодействовать между собой.

Класс и объект: в чем отличие?

*Описанием объекта является **класс**, а **объект** представляет экземпляр этого класса.*

Типы данных встроенные мы прошли ранее.

Класс в C# -это тоже тип данных, только пользовательский тип данных.

```
class Имя_класса { свойства, методы класса и т.д. }
```

```
1.  class Building
2.  {
3.      double width;
4.      double length;
5.      double height;
6.
7.      public double GetVolume()
8.      {
9.          return width * length * height;
10.     }
11. }
```

Конструкторы

Конструкторы вызываются при создании нового объекта и используются для его (объекта) инициализации.

По умолчанию любой класс C# имеет хотя бы один конструктор.

Конструктор по умолчанию:

```
1.  using System;
2.
3.  namespace FirstClass
4.  {
5.      class Program
6.      {
7.          static void Main(string[] args)
8.          {
9.              Building building = new Building();
10.             Console.WriteLine($"Объем здания: {building.GetVolume()}");
11.         }
12.     }
13. }
```

```
Объем здания: 0
```

Собственные конструкторы:

Для класса в C# можно создать любое количество своих конструкторов.
Например, создадим такой конструктор для нашего класса:

```
1.  class Building
2.  {
3.      double width;
4.      double length;
5.      double height;
6.
7.      //Конструктор
8.      public Building(double width, double length, double height)
9.      {
10.         this.width = width;
11.         this.length = length;
12.         this.height = height;
13.     }
14.
15. }
```

название конструктора должно полностью совпадать с названием класса

Наш конструктор должен содержать три обязательных параметра :

```
1.  Building building = new Building(20, 20, 4);
```

Инициализаторы объектов

Теперь посмотрим, как мы можем воспользоваться инициализатором объекта:

```
1.  Building building = new Building { width = 40, length = 30, height = 5 };
```

Здесь мы в фигурных скобках указали значение всем публичным полям класса и без явного вызова конструктора создали объект.

Модификаторы доступа

Всего в C# существует четыре основных ключевых слова для указания уровня доступа :

1.public — доступ к типу или члену класса возможен из любого другого кода в том числе из извне.

2.private — доступ к типу или члену возможен только из кода в том же объекте class или struct.

3.protected — доступ к типу или члену возможен только из кода в том же объекте class либо в class, *производном* от этого class.

4.internal — доступ к типу или члену возможен из любого кода в той же сборке, но не из другой сборки.

Модификаторы доступа можно указывать явно, например, как в нашем классе:

```
1.  public double width;
2.  public double length;
3.  public double height;
```

а можем не указывать, тогда к члену класса по умолчанию будет применен модификатор **private**

Свойства

Почему не стоит давать прямой доступ к полям класса?
Потому что мы можем без проблем сделать вот так:

```
1.  Building building = new Building();
2.  building.height = -100.5;
```

никакой ошибки нет — мы просто задали отрицательную высоту здания и получим в итоге отрицательный объем.

чтобы избежать подобных ситуаций и могут применяться **свойства**.
Свойство позволяет обеспечить контролируемый доступ к полям класса — обеспечить дополнительную логику.

В общем случае, свойство определяется в классе следующим образом:

```
1.  [модификатор_доступа] тип имя_свойства
2.  {
3.    get {возвращаемое_значение;}
4.    set {устанавливаемое_значение;}
5.  }
```

В фигурных скобках содержатся блоки :

get — для чтения поля, которому соответствует свойство

set — для записи поля.

Например, создадим свойство, с помощью которого мы будем определять высоту здания :

```
1.  class Building
2.  {
3.      double height;
4.
5.      public double Height
6.      {
7.          get { return height; }
8.          set
9.          {
10.             if (value < 0)
11.                 throw new Exception("Высота здания не может быть менее 0 метров");
12.             else
13.                 height = value;
14.         }
15.     }
16. }
```

Теперь, если попробовать запустить приложение и выполнить вот такое присваивание:

```
1.  Building building = new Building();
2.  building.Height = -100.5;
```

то программа выдаст ошибку:

```
System.Exception HResult=0x80131500 Сообщение = Высота здания не может быть менее 0 метров Источник = FirstClass
```

мы можем создать свойство «Объем» следующим образом:

```
1.  class Building
2.  {
3.      //метод
4.      private double GetVolume()
5.      {
6.          return width * length * height;
7.      }
8.
9.      //Свойства
10.     public double Volume { get { return GetVolume(); } }
11. }
```

Сокращенная форма записи свойств в C#

Если в блоках get и set свойства не реализуется никакая дополнительная логика, то допускается сокращенная записи свойства.
Например, добавим в наш объект свойство Name (название здания) и применим сокращенную форму записи:

```
1.  class Building
2.  {
3.      public string Name { get; set; }
4.  }
```

В этом случае нам не требуется объявлять в классе дополнительное поле, а блоки get и set остаются пустыми.

Такие свойства также называются *автосвойствами*.

Ключевое слово static

При использовании ключевого слова static метод будет принадлежать именно классу (типу данных), а не объекту и вызвать такой метод можно без создания объекта.

Файл Building.cs

```
1.  class Building
2.  {
3.      static int ordNumber = 0;
4.
5.      static public int GetOrdNumber()
6.      {
7.          return ordNumber;
8.      }
```

Файл Program.cs

```
1.  class Program
2.  {
3.      static void Main(string[] args)
4.      {
5.          Console.WriteLine(Building.GetOrdNumber());//воспользовались статическим методом
```

Статические классы

Объявить статический класс можно так:

```
1.  static class SomeClass
2.   {
3.
4.   }
```

если к классу применяется ключевое слово static, **все члены этого класса должны быть static.**

Где вы можете встретить статический класс C#?

Одним из самых показательных примеров может быть тот самый класс Console

```
5.          Console.WriteLine(Building.GetOrdNumber());
```

Модификатор static позволяет создавать члены класса, не относящиеся к конкретному объекту, а к типу данных (классу) в целом.

С помощью static можно объявлять поля, методы, свойства класса и объявлять целиком статические классы C#.

Часть 2

Простой пример наследования в C#

```
1.  class Building
2.  {
3.      double width;
4.      double length;
5.      double height;
6.      public double GetVolume()
7.      {
8.          return width * length * height;
9.      }
10. }
```

```
1.  class House: Building
2.  {
3.      private int windowCount;
4.      public int WindowCount
5.      {
6.          get => windowCount;
7.          set => windowCount = value > 0 ? value : throw new Exception("Количество");
8.      }
9.  }
```

мы не повторялись в описании нового класса и не переносили в него свойства и методы класса Building — они наследовались в новом классе и мы их можем спокойно использовать:

```
1.  namespace FirstClass
2.  {
3.      class Program
4.      {
5.          static void Main(string[] args)
6.          {
7.              House house = new House
8.              {
9.                  Height = 5,
10.                 Length = 15,
11.                 Width = 20,
12.                 WindowCount = 5
13.             };
14.             Console.WriteLine($"Объем дома {house.Volume}");
15.             Console.WriteLine($"Количество окон {house.WindowCount}");
16.             Console.WriteLine($"Порядковый номер {House.GetOrdNumber()}");
17.             Console.WriteLine($"{house}");
18.         }
19.     }
20. }
```

Пример инкапсуляции

Инкапсуляция — это **когда данные или то, как устроены и работают методы и классы, помещают в виртуальную капсулу, чтобы их нельзя было повредить извне**.
Идея в том, чтобы программист пользовался только теми свойствами и методами, которые есть у класса или объекта доступные для программиста и не лез внутрь.

Пример 1:

```
public double height;
```

Пример 2:

```
private double height;
public double Height
{
    get => height;
    set => height = value > 0 ? value : throw new Exception("Высота здания не может быть менее 0 метров");
}
```

Абстрактный класс

Абстрактные классы, экземпляры которых нельзя создавать.

Создадим абстрактный класс, который будет представлять в программе какого-либо человека:

```
1.  abstract class Person
2.  {
3.      public string Name { get; set; }
4.      public string Family { get; set; }
5.
6.      public Person(string name, string family)
7.      {
8.          Name = name;
9.          Family = family;
10.     }
11.
12.     public string Display()
13.     {
14.         return $"{Family} {Name}";
15.     }
16. }
```

Например, вот такой вызов

```
1.  Person person = new Person("Вася", "Пупкин");
```

приведет к ошибке

Ошибка CS0144 Не удается создать экземпляр абстрактного типа или интерфейса «Person»

Теперь используем наследование и создадим производный от Person класс студента.

```
1.  class Student: Person
2.  {
3.      string Group { get; set; }
4.
5.      public Student(string name, string family, string group) : base(name, family)
6.      {
7.          Group = group;
8.      }
9.  }
```

Здесь мы уже определили свой конструктор для класса и добавили для класса свое свойство — Group (название группы, в которой учится студент).

Теперь, мы можем создать экземпляр этого класса и, например, воспользоваться методом Display:

```
1.  Person student = new Student("Вася", "Пупкин", "ГВН-105");
2.  Student student1 = new Student("Ваня", "Иванов", "ГВН-105");
3.  Console.WriteLine(student.Display());
4.  Console.WriteLine(student1.Display());
```

Абстрактные члены класса

Абстрактные классы в C# могут также иметь **абстрактные члены классов**, которые, также как и класс, определяются с помощью ключевого слова abstract и, при этом, не имеют никакого функционала. Абстрактными могут быть:

- **Методы**
- **Свойства**
- **События**

При использовании абстрактных членов класса следует иметь в виду следующее:

1. абстрактные члены классов не должны иметь модификатор private.
2. производный класс обязан переопределить и реализовать все абстрактные методы и свойства, которые имеются в базовом абстрактном классе. При переопределении в производном классе такой метод или свойство объявляются с модификатором override.
3. если класс имеет хотя бы один абстрактный член, то этот класс должен быть определен как абстрактный.

абстрактные члены класса не должны иметь никакой функциональности,

то есть абстрактный метод в классе должен выглядеть следующим образом:

```
1.  abstract class Person
2.   {
3.       abstract public string Display();
4.   }
```

Интерфейсы в C#

интерфейс — **это контракт** о взаимодействии между различными частями системы. **интерфейс**, в его классическом понимании и представлении, не должен содержать никаких реализаций чего-либо — **только сигнатуры**.

Объявление интерфейса в C#

```
1.  interface IMoneyVault
2.  {
3.      int MoneyAdd(int count);
4.      int MoneyRemove(int count);
5.  }
```

Файл IMoneyVault.cs

наш интерфейс, **во-первых**, не содержит никаких реализаций методов — **только сигнатуры** и, **во-вторых**, методы интерфейса не имеют никаких модификаторов доступа — все они **по умолчанию** публичны и имеют модификатор **public**

Реализация интерфейсов в C#

Файл Person.cs

```
1.  public class Person : IMoneyVault
2.  {
3.      private int currentMoney;
4.
5.      public int MoneyAdd(int count)
6.      {
7.          return currentMoney += count;
8.      }
9.
10.     public int MoneyRemove(int count)
11.     {
12.         return currentMoney -= count;
13.     }
14. }
```

Файл Shop.cs

```
1.  public class Shop : IMoneyVault
2.  {
3.      private int currentMoney;
4.
5.      public int CurrentMoney { get=>currentMoney; }
6.
7.      private double nalog;
8.      public double Nalog { get=>nalog; }
9.
10.
11.     public int MonewAdd(int count)
12.     {
13.         currentMoney += count;
14.         nalog += 0.13 * currentMoney;
15.         return currentMoney;
16.     }
17.
18.     public int MonewRemove(int count)
19.     {
20.         currentMoney -= count;
21.         nalog -= 0.13 * currentMoney;
22.         return currentMoney;
23.     }
24. }
```

Таким образом,
как мы будем реализовывать интерфейс в нашем классе — это наше дело,

но сигнатуры методов и других сущностей объявленных в интерфейсе должны в точности совпадать.

Делегаты в C#

Объявление делегата в C#

Делегат — это специальный объект, который «указывает» на метод или несколько методов. Эти методы могут быть вызваны позднее посредством делегата.

```
1.  delegate void EventHandler();
```

```
1.  delegate double Sum(double x, double y);
```

После того, как мы объявили делегат, он может указывать на любой метод, **имеющий точно такую же сигнатуру**, как и у делегата.

```
namespace Delegates
{
    0 references
    class Program
    {
        1 reference
        delegate void EventHandler();
        1 reference
        delegate double Sum(double x, double y);

        //метод на который будет ссылаться делегат EventHandler
        1 reference
        static void SayHello()
        {
            Console.WriteLine("Hello World!");
        }
        //метод на который будет ссылаться делегат Sum
        1 reference
        static double Calculate(double x, double y)
        {
            return x + y;
        }
        //использование делегатов
        0 references
        static void Main(string[] args)
        {
            EventHandler handler = SayHello;//создаем переменную делегата и присваиваем ей значение
            handler(); //вызываем метод

            Sum sum = Calculate;//создаем переменную делегата и присваиваем ей значение
            double result = sum(15.4, 0.6);//вызываем метод
            Console.WriteLine($"Сумма двух чисел равна {result}");
        }
    }
```

События в C#

Любой класс, и, соответственно, объект этого класса в реальном мире обладает некоторыми свойствами, может совершать некоторые действия, с ним могут происходить некоторые события.

Такая же модель присуща и разработке на языке C#. Однако что это за свойства, действия и события, определяет сам программист, исходя из назначения программы или стоящей перед ним задачи.

Класс «Врач». Что характерно для любого врача?

Свойства	ФИО, должность, дни и часы работы, стаж, зарплата
Методы	пойти на работу, написать заявление на отпуск, выйти из кабинета, обслужить больного, написать научную работу, пообедать
События	заболел, уволен с работы, опоздал на работу, получил стресс, забыл ключи

Класс, который порождает (отправляет) событие, называется *издателем*, а классы, обрабатывающие (принимающие) событие, называются *подписчиками*. Соответственно, на одно и то же событие могут подписываться несколько подписчиков.

В C# событие определяется следующим образом:

```
1.  event delegateType EventName;
```

- **event** — ключевое слово, которое сообщает нам, что перед нами событие
- **delegateType** — это тип делегата. Делегат описывает то, как должен выглядеть метод в подписчике, который будет обрабатывать событие, а также то, какие параметры необходимо передавать подписчику. То есть, делегат, образно выражаясь, представляет из себя некий договор между издателем и подписчиком как они будут между собой общаться.
- **EventName** — название события

Порядок работы с событиями в коде:

1. Объявить делегат, задающий сигнатуру нужного метода. Такой метод будет выполняться при наступлении события в будущем (реагировать на событие);

2. Объявить событие типа ранее созданного делегата;

3. Определить условия, при которых наступает событие;

4. Объявить метод по сигнатуре делегата и подписаться на событие. Теперь этот метод будет реагировать на событие. Этот метод может быть как внутренним, так и во внешнем коде;

5. Дождаться возможного наступления события. Событие может и не наступить при определенных условиях.