



САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭКОНОМИЧЕСКИЙ УНИВЕРСИТЕТ

- **Сотавов Абакар Капланович**
- **Ассистент кафедры Информатики**(наб. канала Грибоедова, 30/32, ауд. 2038)
- e-mail: sotavov@unecon.ru
- Материалы на сайте: <http://de.unecon.ru/course/view.php?id=440>



Иерархии классов



Наследование



Наследование является мощнейшим инструментом ООП. Оно позволяет строить иерархии, в которых классы-потомки получают свойства классов-предков и могут дополнять их или изменять.

Наследование применяется для следующих взаимосвязанных целей:

- исключения из программы повторяющихся фрагментов кода;
- упрощения модификации программы;
- упрощения создания новых программ на основе существующих.

Кроме того, наследование является единственной возможностью использовать классы, исходный код которых недоступен, но которые требуется использовать с изменениями.



[атрибуты] [спецификаторы] **class** имя_класса [: предки]
тело класса

```
class Monster
{ ... // кроме private и public,
  // используется protected
}
class Daemon : Monster
{ ...
}
```

класс - только один
интерфейс - м.б. несколько

Класс в С# может иметь произвольное количество потомков

Класс может наследовать только от одного класса-предка и от произвольного количества интерфейсов.

При наследовании потомок получает [почти] все элементы предка.

Элементы **private** не доступны потомку непосредственно.

Элементы **protected** доступны только потомкам.



класс Monster

```
class Monster {  
    public Monster() // КОНСТРУКТОР  
    {  
        this.name = "Noname";  
        this.health = 100;  
        this.ammo = 100;  
    }  
    public Monster( string name ) : this()  
    { this.name = name; }  
    public Monster( int health, int ammo, string  
name )  
    { this.name = name;  
        this.health = health;  
        this.ammo = ammo;  
    }  
    public int Health { // СВОЙСТВО  
        get { return health; }  
        set { if (value > 0) health = value;  
            else health = 0; }  
    }  
}
```

```
    public int Ammo { // СВОЙСТВО  
        get { return ammo; }  
        set { if (value > 0) ammo = value;  
            else ammo = 0; }  
    }  
    public string Name { // СВОЙСТВО  
        get { return name; }  
    }  
    public void Passport() // МЕТОД  
    { Console.WriteLine(  
        "Monster {0} \t health = {1} \  
ammo = {2}", name, health, ammo );  
    }  
    public override string ToString(){  
        string buf = string.Format(  
            "Monster {0} \t health = {1} \  
ammo = {2}", name, health, ammo);  
        return buf; }  
    string name; // private поля  
    int health, ammo;  
}
```



```
class Daemon : Monster {
    public Daemon() { brain = 1; }
    public Daemon( string name, int brain ) : base( name ) { this.brain = brain; }
    public Daemon( int health, int ammo, string name, int brain )
        : base( health, ammo, name ) { this.brain = brain; }
    new public void Passport() {
        Console.WriteLine( "Daemon {0} \t health ={1} ammo ={2} brain ={3}",
            Name, Health, Ammo, brain );
    }
    public void Think()
    { Console.Write( Name + " is" );
      for ( int i = 0; i < brain; ++i )
          Console.Write( " thinking" );
        Console.WriteLine( "..." );
    }

    int brain;    // закрытое поле
}
```

```
// в классе Monster
public void Passport()
{
    Console.WriteLine(
        "Monster {0} \t health ={1} ammo ={2} name, health",
        Name, Health, Ammo, Name, Health, Ammo );
}
```

```
class Monster {
    public Monster() // КОНСТРУКТОР
    { this.name = "Noname";
      this.health = 100; this.ammo = 100; }
    public Monster( string name )
    { this.name = name; }
    public Monster( int health, int ammo, string name )
    { this.name = name;
      this.health = health;
      this.ammo = ammo; }
}
```



```
public Daemon( string name, int brain ) : base( name )    // 1
{
    this.brain = brain;
}
```

```
public Daemon( int health, int ammo, string name, int brain )
    : base( health, ammo, name )                            // 2
{
    this.brain = brain;
}
```




```
class Class1
{
    static void Main()
    {
        Daemon Dima = new Daemon( "Dima", 3 );    // 5
        Dima.Passport();                          // 6
        Dima.Think();                             // 7
        Dima.Health -= 10;                        // 8
        Dima.Passport();
    }
}
```



Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы (созданные программистом или системой).

Порядок вызова конструкторов:

Если в конструкторе производного класса явный вызов конструктора базового класса отсутствует, автоматически вызывается конструктор базового класса без параметров.

Для иерархии, состоящей из нескольких уровней, конструкторы базовых классов вызываются, начиная с самого верхнего уровня. После этого выполняются конструкторы тех элементов класса, которые являются объектами, в порядке их объявления в классе, а затем исполняется конструктор класса.

Если конструктор базового класса требует указания параметров, он должен быть вызван явным образом в конструкторе производного класса в списке инициализации.



Поля, методы и свойства класса **наследуются**.

При желании **заменить** элемент базового класса новым элементом следует использовать ключевое слово **new**:

// метод класса Daemon (дополнение функций предка)

```
new public void Passport()  
{  
    base.Passport();    // использование функций предка  
    Console.WriteLine( brain );    // дополнение  
}
```

// метод класса Daemon (полная замена)

```
new public void Passport() {  
    Console.WriteLine( "Daemon {0} \  
health = {1} ammo = {2} brain = {3}",  
        Name, Health, Ammo, brain );  
}
```

```
// метод класса Monster  
public void Passport()  
{  
    Console.WriteLine(  
        "Monster {0} \t health = {1} \  
ammo = {2}",  
        name, health, ammo );  
}
```



```
using System;
namespace
    ConsoleApplication1
{ class Monster
    {
        ...
    }
class Daemon : Monster
{
    ... //
}
```

```
class Class1
{
    static void Main()
    { const int n = 3;
      Monster[] stado = new Monster[n];
      stado[0] = new Monster( "Monia" );
      stado[1] = new Monster( "Monk" );
      stado[2] = new Daemon ( "Dimon", 3 );
      foreach ( Monster elem in stado ) elem.Passport(); //
          1
      for ( int i = 0; i < n; ++i ) stado[i].Ammo = 0;
          // 2 Console.WriteLine();
      foreach ( Monster elem in stado ) elem.Passport(); //
          3
    }
}
```



Ссылки разрешаются **до выполнения программы**

Поэтому компилятор может руководствоваться только типом переменной, для которой вызывается метод или свойство. То, что в этой переменной в разные моменты времени могут находиться ссылки на объекты разных типов, компилятор учесть не может.

Поэтому для ссылки базового типа, которой присвоен объект производного типа, можно вызвать только методы и свойства, определенные в базовом классе (т.е. возможность доступа к элементам класса определяется типом ссылки, а не типом объекта, на который она указывает).

```
Monster Vasia = new Daemon();  
Vasia.Passport();
```



- Происходит **на этапе выполнения** программы
- Признак – ключевое слово **virtual** в базовом классе:
virtual public void Passport() ...
- Компилятор формирует для virtual методов *таблицу виртуальных методов*. В нее записываются адреса виртуальных методов (в том числе унаследованных) в порядке описания в классе.

Для каждого класса создается одна таблица.

Связь с таблицей устанавливается при создании объекта с помощью кода, автоматически помещаемого компилятором в конструктор объекта.

Если в производном классе требуется переопределить виртуальный метод, используется ключевое слово **override**:

```
override public void Passport() ...
```

Переопределенный виртуальный метод должен обладать таким же набором параметров, как и одноименный метод базового класса.



```
using System;
namespace ConsoleApplication1
{
class Monster
{
...
virtual public void Passport()
{
Console.WriteLine("Monster {0} \t health = {1}
ammo = {2}", name, health, ammo );
}
...
}
```

```
class Daemon : Monster
{
...
override public void Passport()
{
Console.WriteLine( "Daemon {0} \t
health = {1} ammo = {2} brain =
{3}", Name, Health, Ammo, brain );
}
...
}
```



Абстрактные классы



- Абстрактные классы предназначены для представления **общих понятий**, которые предполагается конкретизировать в производных классах.
- *Абстрактный класс задает интерфейс для всей иерархии.*
- Абстрактный класс задает набор методов, которые каждый из потомков будет реализовывать по-своему.
- Методы абстрактного класса могут *иметь пустое тело* (объявляются как **abstract**).
- Абстрактный класс может содержать и полностью определенные методы (в отличие от интерфейса).
- Если в классе есть хотя бы один абстрактный метод, весь класс должен быть описан как **abstract**.
- Если класс, производный от абстрактного, не переопределяет все абстрактные методы, он также должен описываться как абстрактный.



- Абстрактный класс служит только для порождения потомков.
- Абстрактные классы используются:
 - при работе со структурами данных, предназначенными для хранения объектов одной иерархии
 - в качестве параметров полиморфных методов
- Методу, параметром которого является абстрактный класс, при выполнении программы можно передавать объект любого производного класса.
- Это позволяет создавать *полиморфные методы*, работающие с объектом любого типа в пределах одной иерархии.



Ключевое слово **sealed** позволяет описать класс, от которого, в противоположность абстрактному, наследовать запрещается:

```
sealed class Spirit { ... }  
// class Monster : Spirit { ... }      ошибка!
```

Большинство встроенных типов данных описано как **sealed**. Если необходимо использовать функциональность бесплодного класса, применяется не наследование, а *вложение*, или *включение*: в классе описывается поле соответствующего типа.

Поскольку поля класса обычно закрыты, описывают метод объемлющего класса, из которого вызывается метод включенного класса. Такой способ взаимоотношений классов известен как *модель включения-делегирования* (об этом – далее).



```
abstract class Spirit
{
public abstract void Passport(); }
class Monster : Spirit
{
...
override public void Passport()
{
Console.WriteLine( "Monster {0} \t
health = {1} ammo = {2}", name,
health, ammo );
}
...
}
```

```
class Daemon : Monster
{
...
override public void Passport()
{
Console.WriteLine( "Daemon {0} \t
health = {1} ammo = {2} brain = {3}",
Name, Health, Ammo, brain );
}
...
}
```





СПАСИБО ЗА ВНИМАНИЕ!



ОБЪЕДИНЯЯ ЛУЧШЕЕ