# Наследование в C#

Механизм наследования в C#

# Наследование - мощный инструмент ООП.

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.

# Синтаксис.

**[ атрибуты ] [ спецификаторы ] class имя_класса [ : базовый класс (ы) ]**
**{тело класса}**

**Пример:**

```
class Base
    {  ... // кроме private и public,
         // используется protected
    }
class Derived : Base
    {   ...
    }
```

- Класс в C# 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.

# Пример

```
        class First
           {
         public First(int a int b)
              {   }
}
        class Second : First
           {   public Second(int c, int d) {   }
           }
```

# Конструкторы и наследование

**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

# Вызов конструктора базового класса

```
class First
    {
  public First(int a int b)
        {         }
 }
class Second : First
    {   public Second(): base (5,10)
  {         }
public Second(int c, int d): base(c,d)
  {         }
    }
```

- Базовый класс может иметь несколько конструкторов, каждый из которых может быть вызван с помощью **ключевого слова base** ( при этом вызывается тот конструктор, сигнатура которого соответствует списку аргументов)
- Ключевое слово base  всегда ссылается на базовый класс, находящийся в иерархии наследования непосредственно над вызывающим классом.(если не указывать ключевое слово base, то автоматически вызовется конструктор по умолчанию)

# Ключевое слово this

Из конструктора класса, кроме конструктора базового класса может быть вызван и другой конструктор данного класса, для этого используется ключевое слово this.

```
 public Second (int c, int d, int e):  this (c,d)
{
     e=20;
}
```

Синтаксис аналогичен ключевому слову base

Одновременное использование обоих ключевых слов недопустимо.

# Запечатанный класс

Можно создать класс наследование от которого невозможно.

```
 public sealed SealedClass
{

}
```

При попытке построить класс наследник, возникнет ошибка. Полезно использовать для класса имеющего статические методы.

protected – модификатор действие которого проявляется при наследовании.
Рекомендуется использовать только там, где это необходимо, т. к. он может нарушить инкапсуляцию.

# Модификатор - new

Если в производном классе определить
член класса
 (поля, свойство, метод и т.д.)_
с тем же именем что и в базовом классе,
в этом случае член базового класса
скрывается
 в производном классе.
 Компилятор выдает соответствующее
сообщение
что имя скрывается.
Если программист скрывает член класса
намеренно,
 то он должен указать
ключевое слово  new,
чтобы избежать такого сообщения.

```
namespace CLec_2_cource
{    class Base
    {
        private int a;
        public Base()
        {
            a = 10;
         }
        public Base(int _a)
        {
            a = _a;
        }
       public void Print()
        {
            Console.WriteLine("a=", a);
        }
    }
```

```
class Derived:Base
  {
     private int d;
     public Derived()
     {
        d = 1;
     }
     public  Derived(int _d)
     {
       d= _d;
     }
     public void Print()
     {
  Console.WriteLine(" d={0}", d);
     }
  }
```

```
class Program
    {
        static void Main(string[] args)
        {
            Base ob_b=new Base();
            Derived ob_d=new Derived(5);
            ob_b.Print();
            ob_d.Print();
        }
    }
```

'CLec_2_cource.Derived.Print()' hides inherited member 'CLec_2_cource.Base.Print()'. Use the new keyword if hiding was intended.

```
a=
 d=5
Для продолжения нажмите любую клавишу . . .
```

# Модификатор - new

Используя new – перекрываем соответствующий
 компонент из базового класса.

```
public class Base
{ …………
}
public class Derived:Base
{…………
     new   public void Print()
        {
Console.WriteLine(" d={0}", d);
        }
}
```

Результат работы программы будет тот же,
но исчезнет предупреждающее

Для класса в языке C# возможно использование двух модификаторов доступа:
public - определяет, что нет ограничений на использование класса;
internal - определяет, что класс будет доступен для файлов, входящих в ту же сборку.
**Сборка** - это физический файл, который состоит из нескольких PE-файлов (portable executable), генерируемых компилятором среды .NET. В сборку входит декларация (manifest), содержащая описание сборки для управляющей среды .NET.

Класс может иметь один из следующих модификаторов класса:

`abstract` – определяет, что класс должен быть использован только как базовый класс других классов. Такие классы называются **абстрактными классами** ;

`sealed` – определяет, что класс нельзя использовать в качестве базового класса.

Такие классы в языке C# иногда называются **изолированными классами.**

Очевидно, что изолированный класс не может быть одновременно и абстрактным классом.

Объявление класса может иметь следующее формальное описание:

МодификаторДоступа МодификаторКласса class ИмяКласса : ИмяНаследуемогоКласса {ТелоКласса }

Объектно-ориентированных подход в программирования опирается на следующие принципы:

- инкапсуляция;
- наследование;
- полиморфизм.

**Инкапсуляция** - способность прятать детали реализации объектов от пользователей этих объектов. Инкапсуляция, с одной стороны, позволяет скрывать от пользователя объекта детали его внутренней реализации (принцип "черного ящика"), а, с другой стороны, используется для предотвращения неконтролируемого доступа к внутренним данным объекта.

**Наследование -** возможность создавать новые определения классов на основе существующих, расширяя и переопределяя их функциональность. Наследование используется для повторного использования кода. Наследование бывает двух видов: отношение типа "быть" (классическое наследование) и отношение типа "иметь" (включение или делегирование).
**Полиморфизм** - поддержка выполнения нужного действия в зависимости от типа передаваемого объекта. Полиморфизм применяется для универсальной обработки схожих объектов разных типов. Полиморфизм бывает двух видов: основанный на "раннем связывании" (использует механизм виртуальных методов в классах, связанных классическим наследованием) и "позднем связывании" (используется для объектов, не связанных наследованием).

**Для реализации инкапсуляции в языке C#** используются модификаторы доступа и свойства. Язык C# поддерживает следующие модификаторы доступа (модификаторы видимости):
public - поля, свойства и методы являются общедоступными.
private - поля, свойства и методы будут доступны только в классе, в котором они определены.
protected - поля, свойства и методы будут доступны как в классе, в котором они определены, так и в любом производном класса.
internal - поля, свойства и методы будут доступны во всех классах внутри сборки, в которой определен класс.
Для контроля доступа к полям класса в языке C# можно использовать свойства. Внутри класса свойство определяется в виде пары методов для присвоения значения свойств и для чтения значения свойства. Для пользователей объектов класса свойство представляется как поле класса.

```
тип свойство {
get {
return значение;
}
set {
поле=value;
}
}
```

# Совместимость типов при наследовании

Объекту базового класса можно присвоить объект производного класса:

**предок** ← **потомок**

Это делается для единообразной работы со всей иерархией.

При преобразовании программы из исходного кода в исполняемый используется **два механизма связывания**:

раннее – **early binding** – до выполнения программы

позднее (динамическое) – **late binding** – во время выполнения

# Полиморфизм

Для реализации полиморфизма, основанного на виртуальных методах, в языке C# используются ключевые слова virtual и override:

```
class родитель {
virtual метод(аргументы){
}
}
class потомок: родитель {
override метод(аргументы){
}
}
```

# Виртуальные методы

Виртуальные методы объявляются в базовом классе с ключевым словом virtual, а в производном классе могут быть переопределены. Метод, который переопределяет виртуальный, указывается ключевым словом override. Прототипы виртуальных методов как в базовом, так и в производном классе должны быть одинаковы. Применение виртуальных методов позволяет реализовывать **механизм позднего связывания**.
На этапе компиляции строится только таблица виртуальных методов, а конкретный адрес проставляется уже на этапе выполнения.

При вызове метода - члена класса действуют следующие правила:
для виртуального метода вызывается метод, соответствующий типу объекта, на который имеется ссылка;
для не виртуального метода вызывается метод, соответствующий типу самой ссылки.
При позднем связывании определение вызываемого метода происходит на этапе выполнения (а не при компиляции) в зависимости от типа объекта, для которого вызывается виртуальный метод.
При раннем связывании определение вызываемого метода происходит на этапе компиляции.

# Запрет переопределения методов

Можно запретить переопределение методов и свойств. В этом случает их надо объявить с модификатором sealed

```
class Person
{
….
public virtual void Display()
{….
}
}
class Employee:Person{
….
public override sealed void Display()
{
….
}
```

Далее этот метод не может быть определен в производных классах.

# Пример

```
class AnimalsCat
    {
        public virtual void MakeSound() // если не применить override
        {
            Console.WriteLine("Мяу!");  // тигр скажет «Мяу!»
        }
    }
    class Tiger : AnimalsCat
    {

        public override void MakeSound()
        {
            Console.WriteLine("P-P-P!");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            AnimalsCat n = new Tiger();
            n.MakeSound();
        }
```

```
P-P-P!
Для продолжения нажмите любую клавишу . . . _
```

Если вы объявляете виртуальный метод, то у классов-наследников есть три опции:

- Не реализовывать этот метод
- Реализовывать виртуальную перегрузку этого метода
- Снять виртуальность с этого метода и создать еще один метод с той же сигнатурой

А если мы  не хотим давать возможность делать виртуальную перегрузку метода?

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace CLec_2_cource
{
    class Person
    {
        protected string lastName;
        public Person(string _lastName)
        {
            lastName = _lastName;
        }
        public virtual void Show()
        {
            Console.WriteLine("Я господин: {0}",lastName);
        }
    }
```

```
class Businessman : Person
{
 private string firma;
 public Businessman(string _lastName, string _firma)
        : base(_lastName)
    {
        this.firma = _firma;
    }

public new void Show()
 {
    Console.WriteLine("Я бизнесмен {0}, у меня фирма: {1}",
lastName, firma);
    }
 }
// new скрывает метод базового класса как объявленный с
//virtual так и без этого ключевого слова.
```

```
class Program
    {
        static void Main(string[] args)
        {
            Person p;
            Businessman b = new Businessman("Petroff", "SantaBarbara");
            p = b;// объекту базового класса присвоили ссылку на
производный
            // класс - можно!
            p.Show();
            b.Show();
        }
    }
}
```

Если мы не хотим давать виртуальную перегрузку метода, то используем ключевое слово new:

```
Я господин: Petroff
Я бизнесмен Petroff, у меня фирма: SantaBarbara
Для продолжения нажмите любую клавишу . . .
```

Если мы перегружаем метод, используя override:

```
Я бизнесмен Petroff, у меня фирма: SantaBarbara
Я бизнесмен Petroff, у меня фирма: SantaBarbara
Для продолжения нажмите любую клавишу . . .
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Lecture_VAb
{
     class Employee
   {      protected string firstName;
          protected string lastName;
          protected int age;
          protected double payRate;
          public Employee(string firstName, string lastName, int age, double payRate)
          {
             this.firstName = firstName;
             this.lastName = lastName;
             this.age = age;
             this.payRate = payRate;
          }
          public virtual double CalculatePay(int hoursWorked)
          {
             Console.WriteLine("EmployeeCalculatePay");
             return 50;
          }
   }
        class SalariedEmployee : Employee
        {
          public SalariedEmployee(string firstName, string lastName, int age,
             double payRate):base (firstName,lastName,age,payRate)
          {
          }
          public override double CalculatePay(int hoursWorked)
          {
             Console.WriteLine("EmployeeCalculatePay");
             return 50;
          }
        }




    class ConstructordEmployee : Employee
    {
       public ConstructordEmployee(string firstName, string lastName, int age,
          double payRate)
          : base(firstName, lastName, age, payRate)
       {
       }
       public override double CalculatePay(int hoursWorked)
       {
          Console.WriteLine("ConstructordEmployee");
          return 50;
          //return base.CalculatePay(hoursWorked);
       }
    }
    class HourlyEmployee : Employee
    {
       public HourlyEmployee(string firstName, string lastName, int age,
          double payRate)
          : base(firstName, lastName, age, payRate)
       {
       }
       public override double CalculatePay(int hoursWorked)
       {
          Console.WriteLine("HourlyEmployee");
          return 50;
          //return base.CalculatePay(hoursWorked);
       }
    }
    class DemoPolymorphism
    {
       protected Employee[] epmloyees;
       public void LoadEmployees()
       {
          Console.WriteLine("Загрузка информации о сотрудниках...");
          // в реальном приложении из файла или из БД
          epmloyees = new Employee[3];
          epmloyees[0] = new SalariedEmployee("Jane","Anderson",25,100);
          epmloyees[1] = new ConstructordEmployee("Mike", "Cruise", 35, 120);
          epmloyees[2] = new HourlyEmployee("Martine", "Maison", 45,5);
       }
       public void CalculatePay()
       {
          foreach (Employee emp in epmloyees)
          {
             emp.CalculatePay(50);
          }
       }
    }
class Program
{
   static void Main(string[] args)
   {
      DemoPolymorphism demo = new DemoPolymorphism();
      demo.LoadEmployees();
      demo.CalculatePay();
   }   }}
```

# Наследование- итоги

Итоги:

**Наследование можно рассматривать как средство усложнения базового класса.**

Класс потомок будет иметь:

Унаследованные поля, методы и свойства

Дополнительные поля, методы и свойства

Конструкторы не наследуются!

Конструктор потомка не имеет вызова конструктора предка,

Автоматически вызывается конструктор предка без параметров

Конструктор потомка явно вызывает конструктор предка

**Наследование – средство изменения базового класса.**

Реализуется через переопределение полей, методов и свойств базового класса в потомке.  На практике это чаще изменение поведения – т.е. переопределение методов.

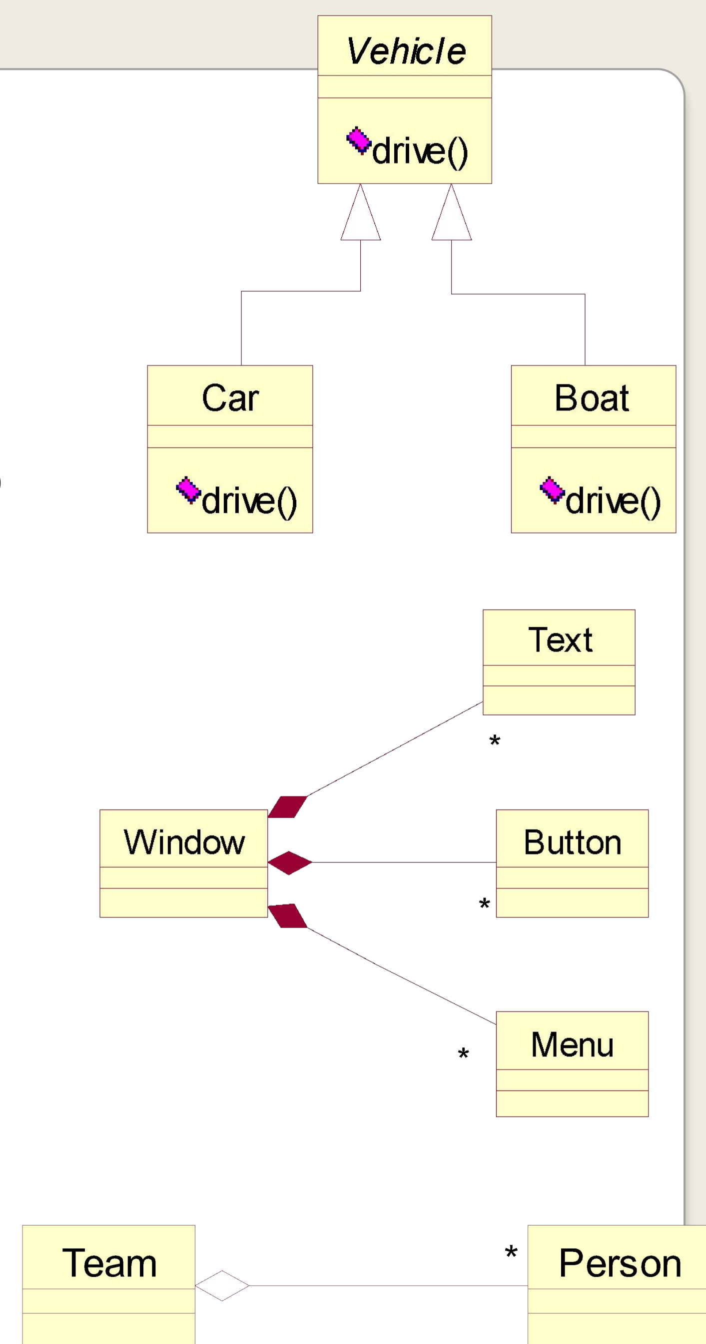
Два способа переопределения методов:

new  и override

1) new – новая версия метода, через ссылку класса-предка вызывается определенный в классе предке, а через ссылку класса потомка, метод определенный в классе –потомке.
2) override – это метод заменяющий метод предка, может вызываться через ссылку класса-потомка или ссылку базового класса.

# Наследование и вложение

- *Наследование* класса Y от класса X чаще всего означает, что Y представляет собой разновидность класса X (более конкретную, частную концепцию).
- *Вложение* является альтернативным наследованию механизмом использования одним классом другого: один класс является полем другого.
- *Вложение* представляет отношения классов «Y содержит X» или «Y реализуется посредством X» и реализуется с помощью модели «включение-делегирование».



Полезно: **super, base, inherited –** используем для обозначения   базового класса
Derived,
Язык C# поддерживает объявление абстрактных методов с помощью ключевого слова abstract. Абстрактные методы не содержат реализации и должны быть переопределены с помощью ключевого слова override в классах-потомках. Класс, в котором объявлен хотя бы один абстрактный метод, является абстрактным и должен быть объявлен в помощью ключевого слова abstract. Объекты абстрактного класса не могут быть созданы.

# Абстрактные классы

Абстрактным классом называется класс, который содержит один или несколько *абстрактных методов*.
Абстрактный класс не может использоваться для создания объектов.
Как правило, абстрактный класс описывает некий интерфейс, который должен быть реализован всеми его производными классами.
Абстрактный метод языка C# не имеет тела метода и аналогичен *чисто виртуальному методу* языка C++.

Например:

```
public abstract int M1(int a, int b);
```

**Абстрактный класс** можно использовать только как базовый для других классов. При этом если производный класс не содержит реализации *абстрактного метода*, то он также является абстрактным классом.

По умолчанию при создании абстрактного класса в среде **VisualStudio .NET** в формируемый абстрактный класс автоматически вставляется только один метод – конструктор без параметров.

В приведенном ранее примере класс Employee представлял просто служащего, в реальности такого объекта быть не должно, т.к. объект каждой из возможных категорий сотрудников можно вполне в соответствие дать производный класс.
Абстрактные методы – аналоги чисто виртуальных функций в C++.
Рассмотрим пример:
Абстрактный класс Фигура и производные от него классы:
Треугольник и Прямоугольник

```
public   abstract   class Shape
{
        protected abstract double getArea();
// переопределяемый абстрактный метод
    public void Print() // рекурсивный метод
        {      Console.WriteLine("Площадь фигуры="+getArea());
        }
        }
```

```
public class Triangle : Shape
  {   private double a;
      private double b;
      private double c;
      public Triangle()
      {       }
      public Triangle(double a, double b, double c)
      {  this.a = a;
         this.b = b;
         this.c = c;
      }
```

```
protected override double getArea()
    {   double p = (a + b + c) / 2.0;
        return Math.Sqrt((p-a)*(p-b)*(p-c)); ;
    }
  }
```

```
public class Rectangle : Shape
  {   private double a;
      private double b;
      public Rectangle(double a, double b)
      {
         this.a = a;
         this.b = b;

      }
       protected override double getArea()
      {
              return a*b ;
      }
  }
```

```
class Program
  {  static void Main(string[] args)
     {
         Triangle obT = new Triangle(3,3,3);

         Rectangle obR = new Rectangle(1,2.5);
         Console.WriteLine("Площадь треугольника");
         obT.Print();
         Console.WriteLine("Площадь прямоугольника");
         obR.Print();
     }
  }
```

# Рекомендации по программированию

- Главная цель, к которой нужно стремиться, — получить легко читаемую программу возможно более простой структуры.
- Создание программы надо начинать с определения ее исходных данных и результатов.
- Следующий шаг — записать на естественном языке (возможно, с применением обобщенных блок-схем), что именно и как должна делать программа.
- При кодировании необходимо помнить о принципах структурного программирования: программа должна состоять из четкой последовательности блоков — базовых конструкций.
- Имена переменных должны отражать их смысл. Переменные желательно инициализировать при их объявлении
- Следует избегать использования в программе чисел в явном виде.
- Программа должна быть «прозрачна». Для записи каждого фрагмента алгоритма необходимо использовать наиболее подходящие средства языка.

- В программе полезно предусматривать реакцию на неверные входные параметры.
- Необходимо предусматривать печать сообщений или выбрасывание исключения в тех точках программы, куда управление при нормальной работе программы передаваться не должно.
- Сообщение об ошибке должно быть информативным и подсказывать пользователю, как ее исправить.
- После написания программу следует тщательно отредактировать
- Комментарии должны представлять собой правильные предложения без сокращений и со знаками препинания
- Вложенные блоки должны иметь отступ в 3–5 символов

Форматируйте текст по столбцам везде, где это возможно:

```
    string   buf       = "qwerty";
    double  ex         = 3.1234;
    int       number = 12;
    byte      z          = 0;
...
    if ( done ) Console.WriteLine( "Сумма ряда - " + y );
    else          Console.WriteLine( "Ряд расходится" );
    ...
    if       ( x >= 0 && x < 10 ) y = t * x;
    else if ( x >= 10 )             y = 2 * t;
    else                                 y = x;
```

После знаков препинания должны использоваться пробелы:

```
 f=a+b;                     // плохо! Лучше f = a + b;
```

"Вопрос «Как писать хорошие программы на С++?» напоминает вопрос «Как писать хорошую английскую прозу?». Есть два совета: «Знай, что хочешь сказать» и «Тренируйся. Подражай хорошему стилю». Оба совета годятся как для С++, так и для английской прозы, и обоим одинаково сложно следовать."

Б. Страуструп

# К чему следует стремиться?

http://ips.ifmo.ru/information/ind
ex.html