# **Курс** «C#. Программирование на языке высокого уровня»

Павловская Т.А.

# Лекция 8. Наследование классов

**Организация иерархий классов. Раннее и позднее связывание. Виртуальные методы. Абстрактные и бесплодные классы. Виды взаимоотношений между классами.**

# Возможности наследования

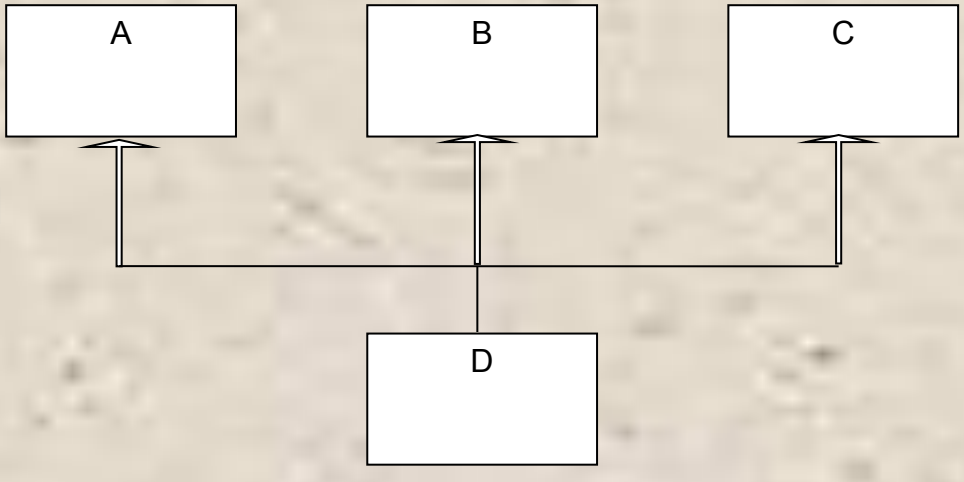
- Наследование является мощнейшим инструментом ООП. Оно позволяет строить иерархии, в которых классы-потомки получают свойства классов-предков и могут дополнять их или изменять.
- Наследование применяется для следующих взаимосвязанных целей:
  - исключения из программы повторяющихся фрагментов кода;
  - упрощения модификации программы;
  - упрощения создания новых программ на основе существующих.
- Кроме того, наследование является единственной возможностью использовать объекты, исходный код которых недоступен, но в которые требуется внести изменения.

# Синтаксис

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

**тело класса**

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




- 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 все элементы предка.
- Элементы private не доступны потомку непосредственно.
- Элементы protected доступны только потомкам.

# Сквозной пример класса

```
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;
}
```

# Daemon, наследник класса Monster

```
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
}
```

```
class Monster {

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

# 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 конструкторе производного класса в списке инициализации.

# Вызов конструктора базового класса

```
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;
        }
```

# Наследование полей и методов

- 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} \t
    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 );
      }
```

# Совместимость типов при наследовании

Объекту базового класса можно присвоить объект производного класса:

предок  потомок

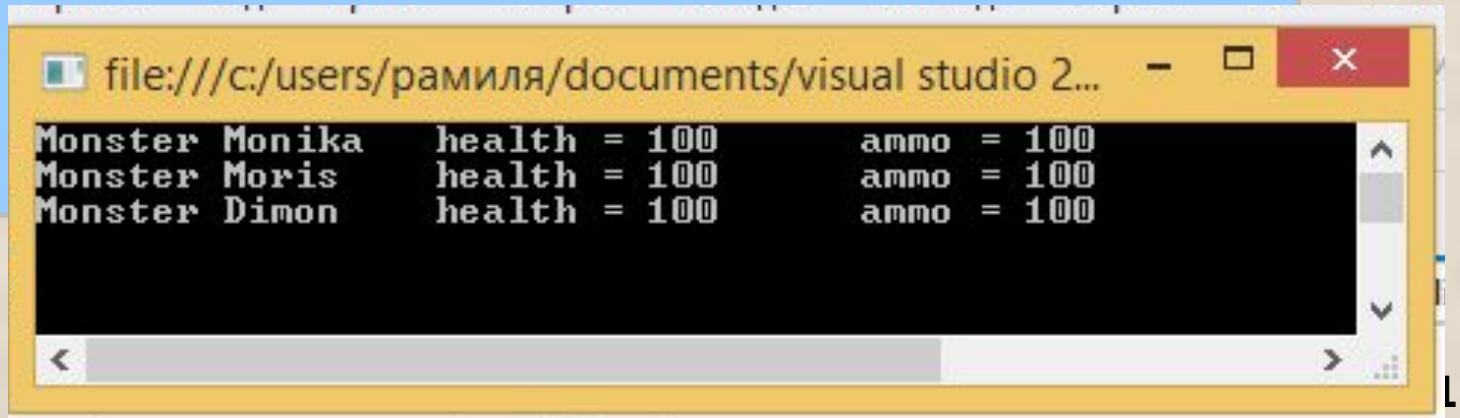
Это делается для единообразной работы со всей иерархией

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

# Пример раннего связывания

```
class Program
  {
     static void Main(string[] args)
     {
        Monster[] stado = new ConsoleApplication22.Monster[3];

        stado[0] = new ConsoleApplication22.Monster("Monika");
        stado[1] = new ConsoleApplication22.Monster("Moris");
        stado[2] = new Daemon("Dimon", 3);

        for(int i=0;i<3;i++) stado[i].Passport();

        Console.ReadKey();
     }
  }
```



# Раннее связывание

- 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).

# Позднее связывание

- Происходит на этапе выполнения программы
- Признак – ключевое слово **virtual** в базовом классе:

```
virtual public void Passport() ...
```

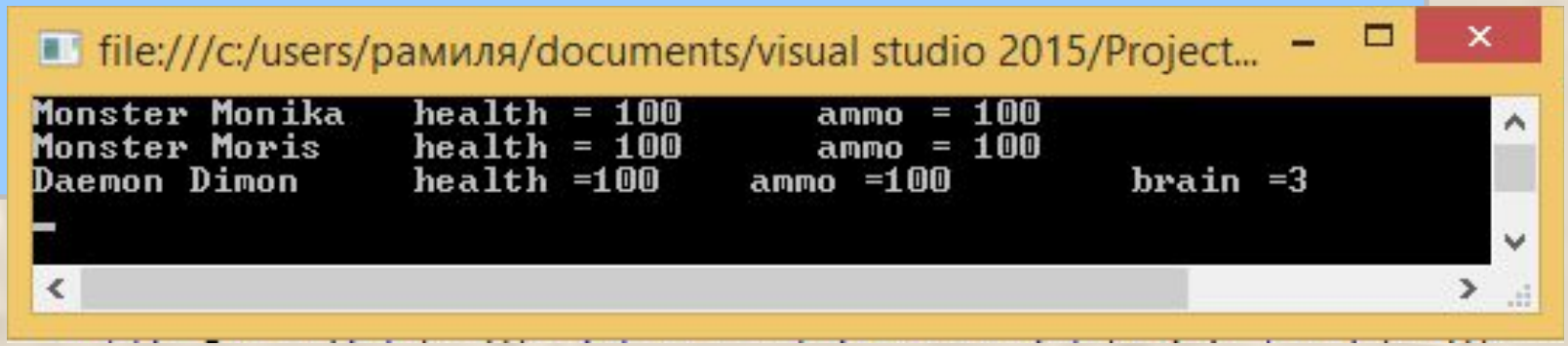
- Компилятор формирует для virtual методов *таблицу виртуальных методов*. В нее записываются адреса виртуальных методов (в том числе унаследованных) в порядке описания в классе. Для каждого класса создается одна таблица.
- Связь с таблицей устанавливается при создании объекта с помощью кода, автоматически 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*.

# Пример позднего связывания

```
class Program
   {
      static void Main(string[] args)
      {
         Monster[] stado = new ConsoleApplication22.Monster[3];

         stado[0] = new ConsoleApplication22.Monster("Monika");
         stado[1] = new ConsoleApplication22.Monster("Moris");
         stado[2] = new Daemon("Dimon", 3);

         for(int i=0;i<3;i++) stado[i].Passport();

         Console.ReadKey();
      }
   }
```



# Полиморфизм

- *Виртуальные методы базового класса определяют интерфейс всей иерархии*.
- Он может расширяться в потомках за счет добавления новых виртуальных методов. Переопределять виртуальный метод в каждом из потомков не обязательно: если он выполняет устраивающие потомка действия, метод наследуется.
- Вызов виртуального метода выполняется так: из объекта берется адрес его таблицы вирт. методов, из нее выбирается адрес метода, а затем управление передается этому методу.
- Таким образом, при использовании виртуальных методов из всех одноименных методов иерархии всегда выбирается тот, который соответствует фактическому типу вызвавшего его объекта.
- С помощью виртуальных методов реализуется один из основных принципов объектно-ориентированного программирования — *полиморфизм*.

# Применение виртуальных методов

- Виртуальные методы используются при работе с производными классами через ссылку на базовый класс
- Виртуальные методы незаменимы также при передаче объектов в методы в качестве параметров. В параметрах метода описывается объект базового типа, а при вызове в нее передается объект производного класса. В этом случае виртуальные методы, вызываемые для объекта из метода, будут соответствовать типу аргумента, а не параметра.

- При описании классов рекомендуется определять в качестве виртуальных те методы, которые в производных классах должны реализовываться по-другому. Если во всех классах иерархии метод будет выполняться одинаково, его лучше определить как обычный метод.

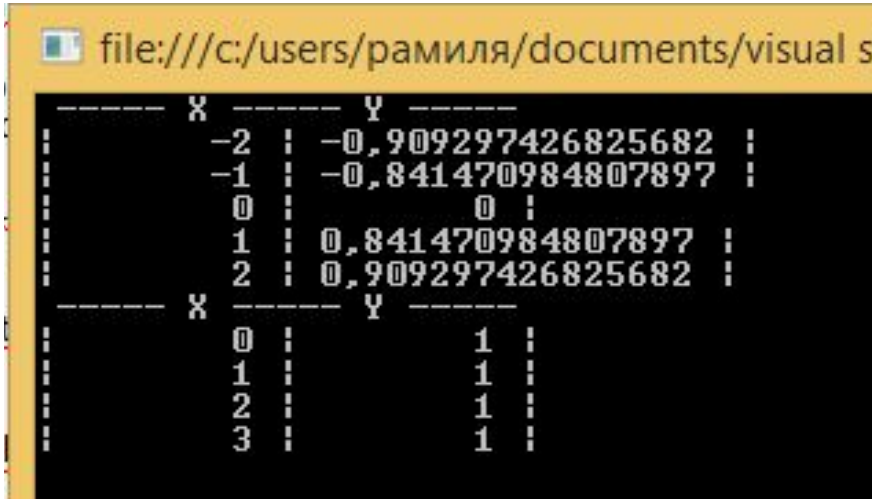
# Абстрактные классы

- *Абстрактный класс служит только для порождения потомков*. Как правило, в нем задается набор методов, которые каждый из потомков будет реализовывать по-своему. Абстрактные классы предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах.
- *Абстрактный класс задает интерфейс для всей иерархии*, при этом методам класса может не соответствовать никаких конкретных действий. В этом случае методы имеют пустое тело и объявляются со спецификатором **abstract**.
- Если в классе есть хотя бы один абстрактный метод, весь класс также должен быть описан как абстрактный (со спецификатором **abstract**).
- Абстрактный класс может содержать и полностью определенные методы, в отличие от интерфейса.

# Полиморфные методы

- Абстрактные классы используются:
  - при работе со структурами данных, предназначенными для хранения объектов одной иерархии
  - в качестве параметров методов.
- Если класс, производный от абстрактного, не переопределяет все абстрактные методы, он также должен описываться как абстрактный.
- Можно создать метод, параметром которого является абстрактный класс. На место этого параметра при выполнении программы может передаваться объект любого производного класса. Это позволяет создавать *полиморфные методы*, работающие с объектом любого типа в пределах одной иерархии.

```
abstract class TableFun
   {
      public abstract double F(double x);
      public void Table(double x,double b)
      {
         Console.WriteLine(" ----- X ----- Y -----");
         while (x <= b)
         {
            Console.WriteLine("| {0} | {1} |", x, F(x));
            x += 1;
         } }    }
   class SimpleFun : TableFun
   {      public override double F(double x)       return 1;


   }


   class SinFun : TableFun
   {
      public override double F(double x)
         return Math.Sin(x);
}
class Program
   {       static void Main()
      {
         TableFun a = new SinFun();
         a.Table(-2, 2);
         a = new SimpleFun();
         a.Table(0, 3);
         Console.ReadKey();
      }   }}
```



# Бесплодные (финальные) классы

- 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 в сл. 26*).

# Класс object

- Корневой класс System.Object всей иерархии объектов .NET, называемый в C# object, обеспечивает всех наследников несколькими важными методами.
- Производные классы могут использовать эти методы непосредственно или переопределять их.

- Класс object используется непосредственно:
  - при описании типа параметров методов для придания им общности;
  - для хранения ссылок на объекты различного типа.

# Открытые методы класса System.Object

public virtual bool Equals(object obj);

- возвращает true, если параметр и вызывающий объект ссылаются на одну и ту же область памяти (для значимых типов определяет, считаются ли равными указанные экземпляры объектов)

public static bool Equals(object ob1, object ob2);

- возвращает true, если оба параметра ссылаются на одну и ту же область памяти

public virtual int GetHashCode();

- формирует хэш-код объекта и возвращает число, однозначно идентифицирующее объект. Каждому объекту Type соответствует своя хэш-функция, у которой для обеспечения уникальности в качестве входного аргумента должно использоваться хотя бы одно из полей экземпляра

public Type GetType();

- возвращает текущий полиморфный тип объекта (не тип ссылки, а тип объекта, на который она в данный момент указывает)

public static bool ReferenceEquals(object ob1, object ob2);

- возвращает true, если оба параметра ссылаются на одну и ту же область памяти

public virtual string ToString()

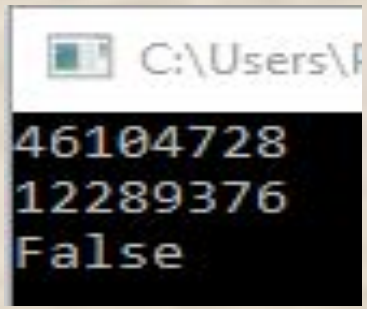
- возвращает для ссылочных типов полное имя класса в виде строки, а для значимых — значение величины, преобразованное в строку. Этот метод переопределяют, чтобы выводить информацию о состоянии объекта.

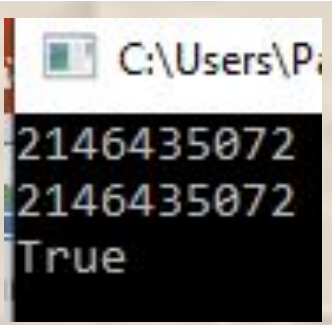
Нужно разделять понятия "идентичность" и "эквивалетность". Идентичность характеризуется адресом объекта в памяти, и любой объект идентичен себе и только себе. Реализация методов GetHashCode и Equals по умолчанию для ссылочных типов проверяет идентичность. Эквивалетность же диктуется обычно предметной областью, в данном случае математикой, которая устанавливает правила сравнения комплексных чисел на равенство.

```
public class Complex
   {
      private double _real;
      private double _imaginary;

      public Complex(double real, double imaginary)
      {
         _real = real;
         _imaginary = imaginary;
      }

      public double Real { get { return _real; } }
      public double Imaginary { get { return _imaginary;
} }
   };
```

```
public override int GetHashCode()
        {
            return _real.GetHashCode() ^
_imaginary.GetHashCode();
        }

        public override bool Equals(Object obj)
        {
            Complex other = obj as Complex;

            if (other == null)
                return false;

            return other._real == _real &&
other._imaginary == _imaginary;
        }
```

```
//создаем два экземпляра класса Complex с
одинаковыми значениями полей

Complex first = new Complex(1.0, 2.0);
        Complex second = new Complex(1.0, 2.0);
        Console.WriteLine(first.GetHashCode());
        Console.WriteLine(second.GetHashCode());
        Console.WriteLine(second.Equals(first));
```



# Пример переопределения метода Equals

```
// сравнение значений, а не ссылок
public override bool Equals( object obj ) {
    if ( obj == null || GetType() != obj.GetType() ) return false;
    Monster temp = (Monster) obj;
    return  health == temp.health &&
            ammo   == temp.ammo   &&
            name   == temp.name;
}
public override int GetHashCode()
{
    return name.GetHashCode();
}
```

# Рекомендации по программированию

- Главное преимущество наследования состоит в том, что на уровне базового класса можно написать универсальный код, с помощью которого работать также с объектами производного класса, что реализуется с помощью виртуальных методов.
- Как виртуальные должны быть описаны методы, которые выполняют во всех классах иерархии одну и ту же функцию, но, возможно, разными способами.
- Для представления общих понятий, которые предполагается конкретизировать в производных классах, используют абстрактные классы. Как правило, в абстрактном классе задается набор методов, то есть интерфейс, который каждый из потомков будет реализовывать по-своему.
- Обычные методы (не виртуальные) переопределять в производных классах не рекомендуется.

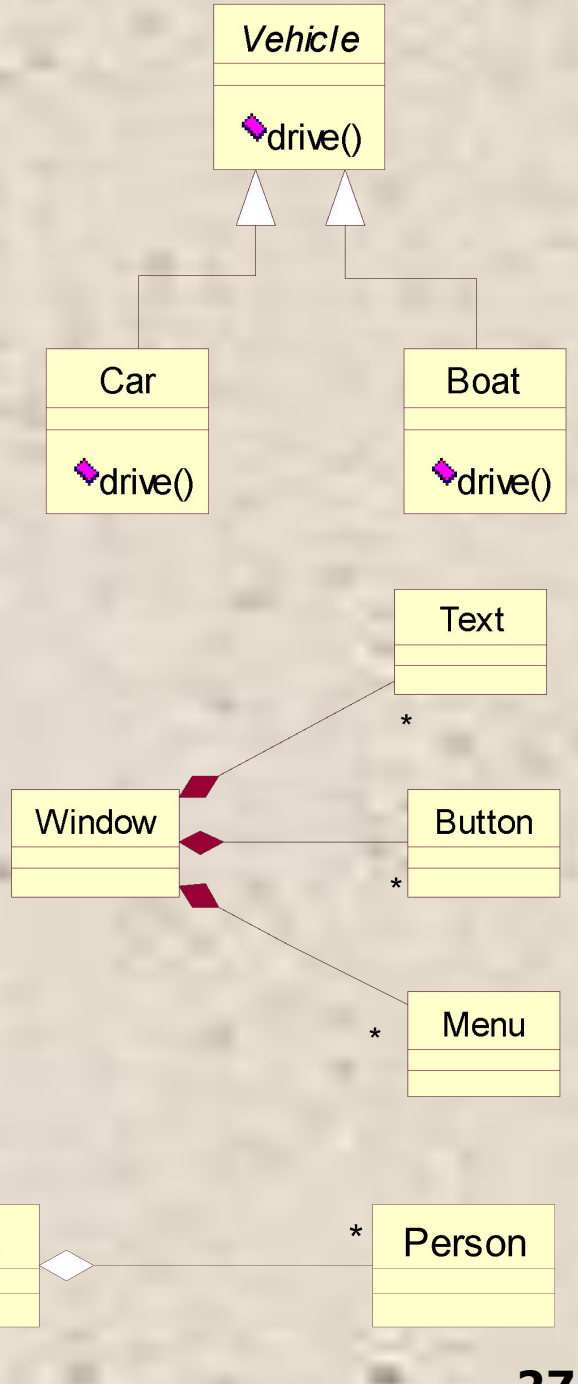
# Виды взаимоотношений между классами

***Классификация Тимоти Бадда***

- Наследование
  - Специализация (Класс-наследник является специализированной формой родительского класса — в наследнике просто переопределяются методы)
  - Спецификация (*Дочерний класс* реализует поведение, описанное в родительском классе. В C# эта форма реализуется наследованием от абстрактного класса)
  - Конструирование или Варьирование(Наследник использует методы предка, но не является его подтипом; предок и потомок являются вариациями на одну тему – например, прямоугольник и квадрат)
  - Расширение (В потомок добавляют новые методы, расширяя поведение предка)
  - Обобщение (Потомок обобщает поведение предка. Обычно такое наследование используется в тех случаях, когда изменить поведение базового класса невозможно (например, базовый класс является библиотечным классом))
  - Ограничение (Потомок ограничивает поведение предка)
- Вложение
  - Композиция (форма агрегирования с четко выраженным отношением владения, причем время жизни частей и целого совпадают))
  - Агрегация (специальная форма ассоциации, которая служит для представления отношения типа "часть-целое" между агрегатом (целое) и его составной частью )

# 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».



# Модель включения-делегирования

```
class Двигатель {public void Запуск() {Console.WriteLine( "вжжж!!" ); }}

class Самолет
   {     public Самолет()
            {    левый  = new Двигатель(); правый = new Двигатель(); }
         public void Запустить_двигатели()
            {     левый.Запуск();  правый.Запуск();    }
         Двигатель левый, правый;
   }
   class Class1
   {   static void Main()
      {        Самолет АН24_1 = new Самолет();
               АН24_1.Запустить_двигатели();
      }
   }
```

Результат работы программы:
вжжж!!
вжжж!!