
Наследование

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 классы, исходный код которых недоступен, но которые требуется использовать с изменениями.

Синтаксис

[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 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.

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

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;
}
```

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

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} \  
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:



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

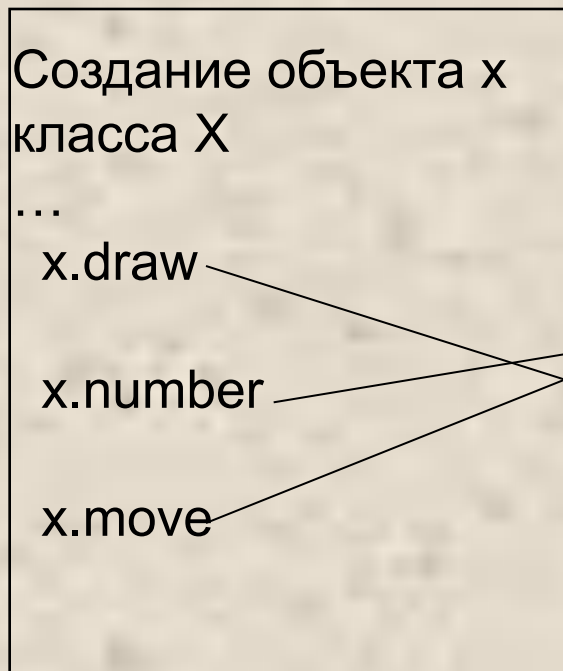
Пример раннего связывания

```
class T {  
    T(int i) { this.i = i; }  
    draw { вывод "ТТ" }  
    erase  
    move { erase, ->, draw }  
    number { вывод i }  
    protected int i;  
}  
class X : T {  
    X(int i) : base(i) {}  
    new draw { вывод "ХХ" }  
    new erase  
    resize  
}  
// все методы public
```

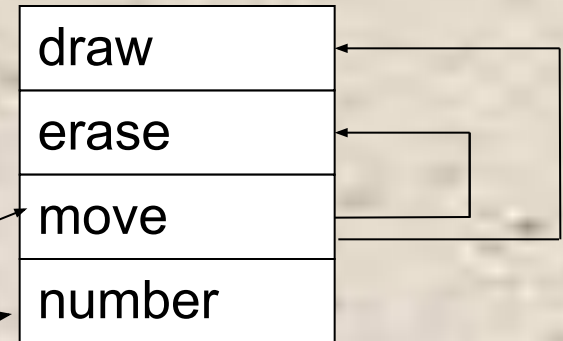
```
// одиночный объект:  
X x = new X(15);  
x.draw(); // ХХ  
x.number(); // 15  
x.move() // ТТ  
// массив объектов баз. типа:  
T mas = new T[n];  
mas[0] = new T(10);  
mas[1] = new T(20);  
mas[2] = new X(15);  
mas[3] = new X(25);  
  
foreach (T t in mas) t.number()  
// 10 20 15 25  
foreach (T t in mas) t.draw()  
// ТТ ТТ ТТ ТТ  
// t.resize – не работает
```

Раннее связывание

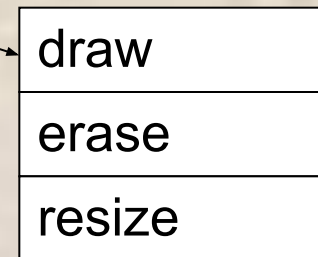
Вызывающий метод



методы класса T



Методы класса X



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).

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

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

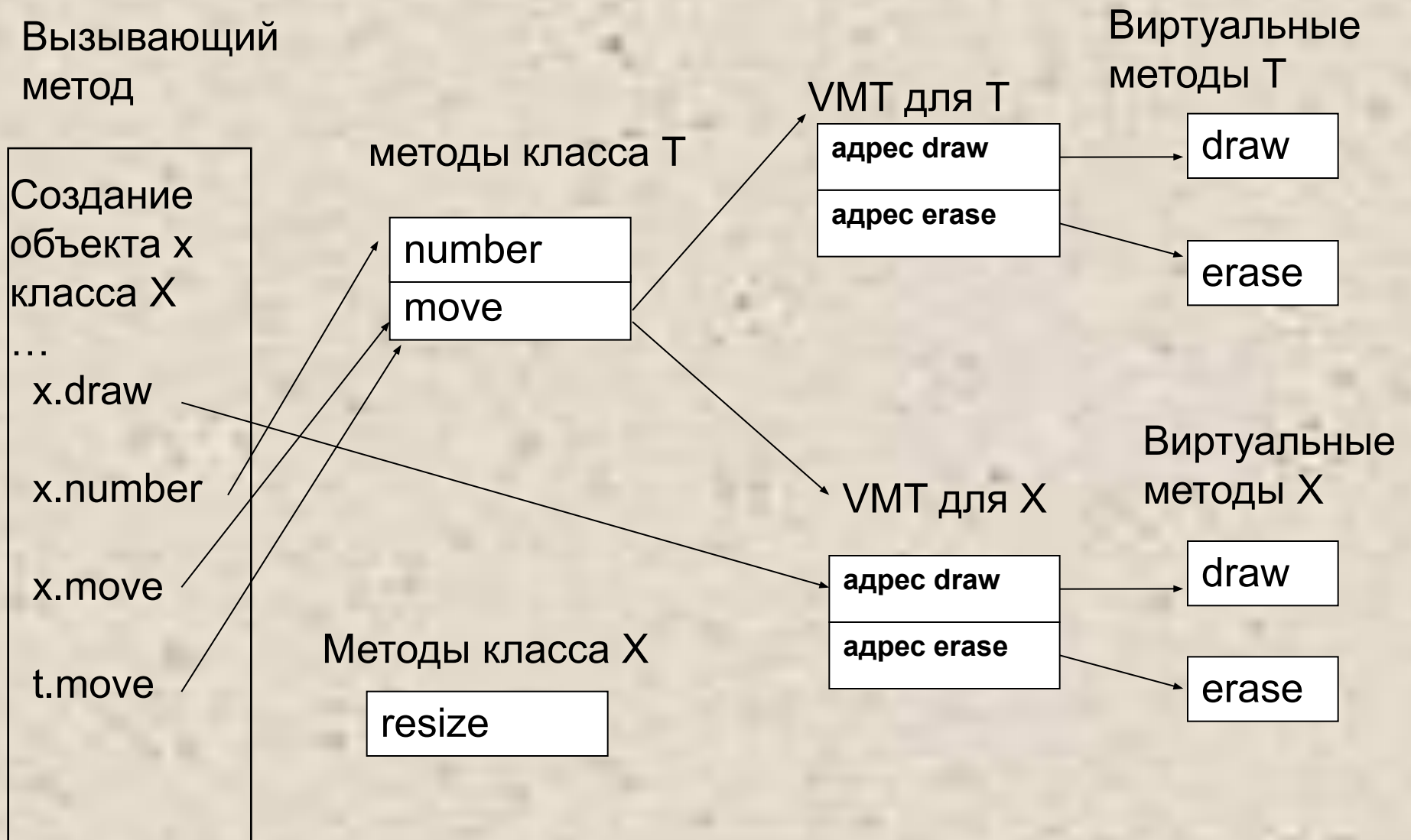
```
class T {
    T(int i)
    virtual draw { "TT" }
    virtual erase
        move { erase, ..., draw }
        number { i }
    protected int i;
}
class X : T {
    X(int i)
    override draw { "XX" }
    override erase
        resize
}

// все методы public
```

```
// одиночный объект:
X x = new X(15);
x.draw();           // XX
x.number();         // 15
x.move()            // XX
// массив объектов баз. типа:
T mas = new T[n];
mas[0] = new T(10);
mas[1] = new T(20);
mas[2] = new X(15);
mas[3] = new X(25);

foreach (T t in mas) t.number()
// 10  20  15  25
foreach (T t in mas) t.draw()
// TT  TT  XX XX
```

Позднее связывание



Полиморфизм

- Виртуальные методы базового класса определяют **интерфейс** всей иерархии.
- Интерфейс может расширяться в потомках за счет добавления новых виртуальных методов (нежелательно).
- Переопределять виртуальный метод в каждом из потомков не обязательно: если он выполняет устраивающие потомка действия, метод наследуется.
- Механизм вызова виртуального метода:
 - из объекта берется адрес таблицы вирт. методов соотв. класса
 - из нее выбирается адрес метода
 - ему передается управление.
- Итак, при использовании виртуальных методов из всех одноименных методов иерархии всегда выбирается тот, который **соответствует фактическому типу** вызвавшего его объекта.
- Виртуальные методы — основное проявление *полиморфизма*.

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


Применение виртуальных методов

- Виртуальные методы используются:
 - при работе с производными классами через ссылку на базовый класс;
 - при передаче объектов в методы в качестве параметров:

В параметрах метода описывается объект базового типа, а при вызове в нее передается объект производного класса. Виртуальные методы, вызываемые для параметра метода, будут соответствовать типу аргумента, а не параметра.

Методы, работающие с полиморфными объектами, называют полиморфными.

Пример полиморфного метода

```
КакойтоМетод ( T t
)
{
    t.draw(); ...
}
```

- T - предок
- X - потомок
- в обоих классах есть виртуальный метод draw
- T t = new T(10);
- X x = new X(20);
- КакойтоМетод(t) вызывается draw из T
- КакойтоМетод(x) вызывается draw из X

- 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

- 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.

Применение абстрактных классов

- *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.

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).

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

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

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, для значимых — значение величины, преобразованное в строку. Этот метод переопределяют, чтобы выводить информацию о состоянии объекта.

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

■ Наследование

- Специализация (Наследник является специализированной формой предка)
- Спецификация (Дочерний класс реализует поведение, описанное в предке)
- 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)
- Ограничение (Потомок ограничивает поведение предка)

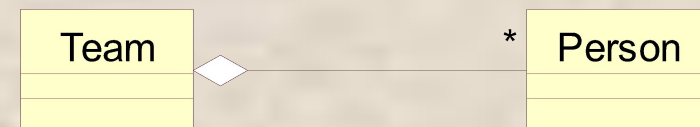
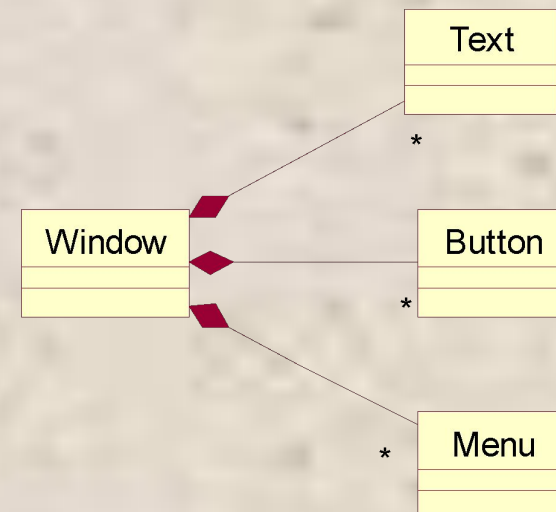
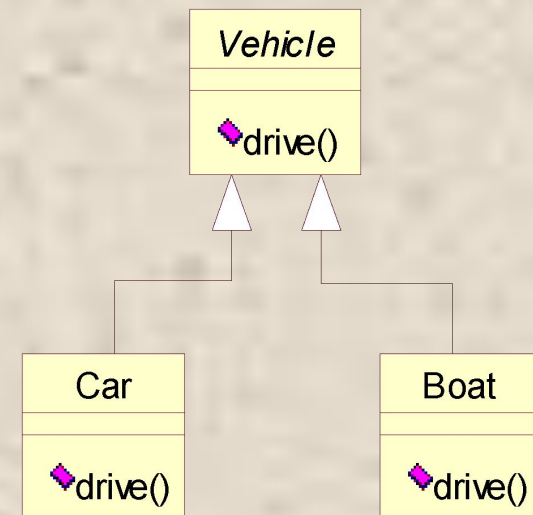
■ Вложение

- композиция
- агрегация

Классификация Тимоти Бадда

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!!
```