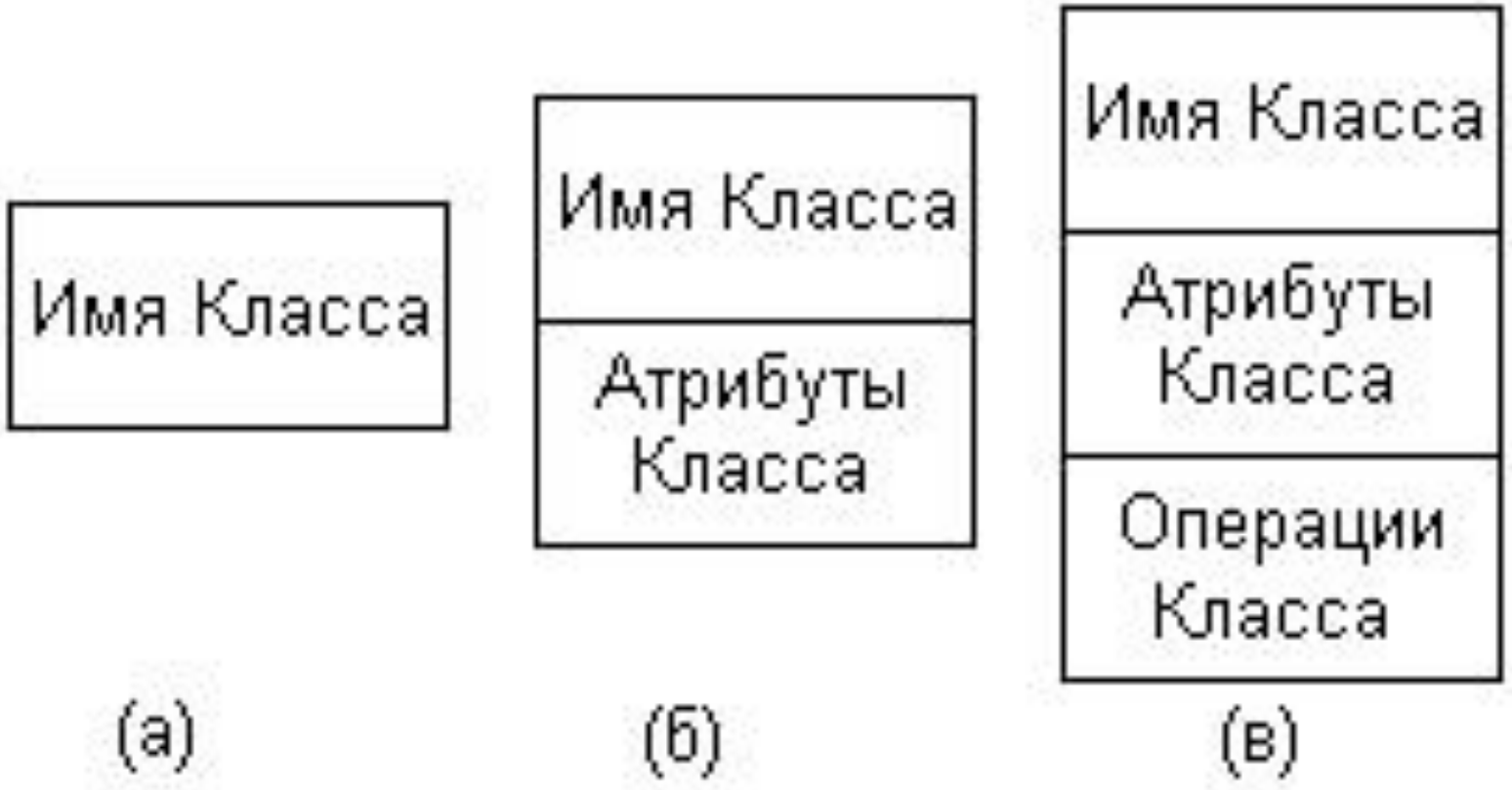
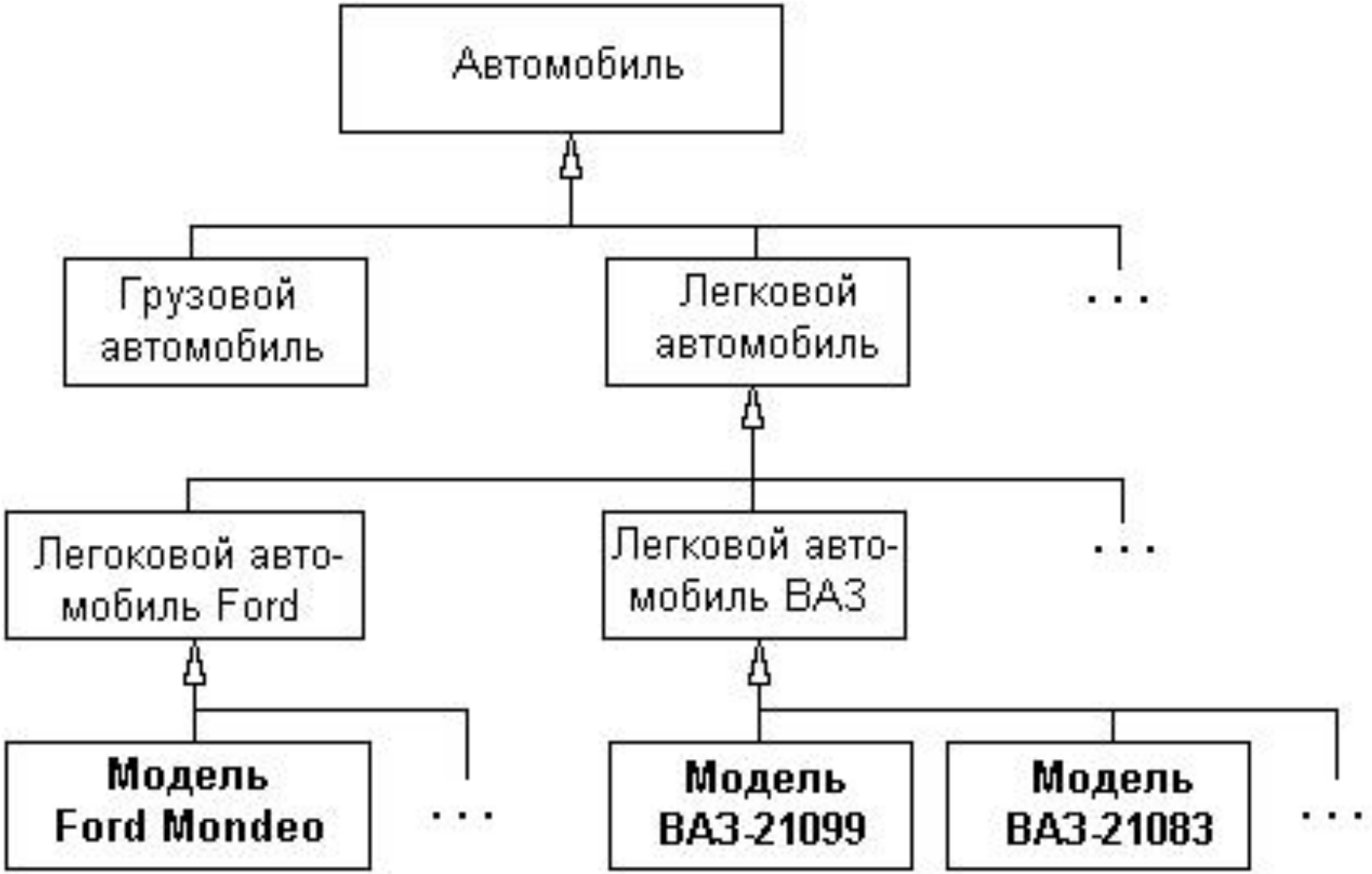
Наследование



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 объекты**, **исходный код** которых **недоступен**, но в которые требуется внести изменения.


Автомобиль
Грузовой автомобиль
Легковой автомобиль
. . .
Легоковой авто-мобиль Ford
Легковой авто-мобиль ВАЗ
. . .
Модель Ford Mondeo
. . .
Модель ВАЗ-21099
Модель ВАЗ-21083
. . .

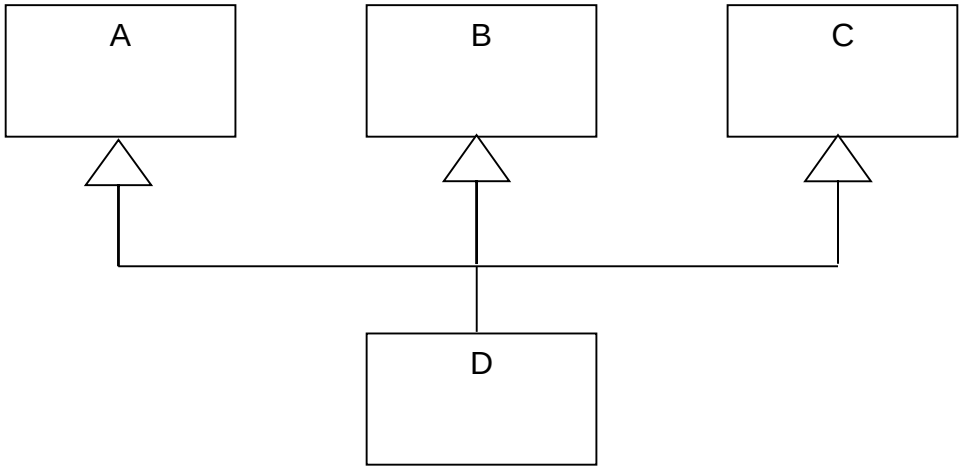
Синтаксис

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

тело класса

- Класс в C# может иметь произвольное количество потомков и только один класс-предок.
- Класс может наследовать от произвольного количества интерфейсов.

```
class Monster
{
    ...
}

class Daemon : Monster
{
        ...
}
```



Конструкторы и наследование

Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы.

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 явным образом вызван в конструкторе производного класса в списке инициализации.

Еще раз: пример описания 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 GetName()          // метод
    {  return name;   }
  public int GetAmmo()         // метод
     {  return ammo;}
```

```
public int Health {                  // свойство
   get { return health;  }
   set { if (value > 0) health = value;
           else                health = 0;
         }
 }
 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;  }

  private string name;
  private 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
}}
```

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 явным образом это указать с помощью ключевого слова new:

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

- Элементы базового класса, определенные как private, в производном классе недоступны.

Раннее связывание

- объекту базового класса можно присвоить объект производного класса
- для него вызываются 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).
- это объясняется тем, что ссылки разрешаются до выполнения программы (ранее связывание)
- 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.

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* (Virtual Method Table, VMT). В нее записываются адреса виртуальных методов (в том числе унаследованных) в порядке описания в классе. Для каждого класса создается одна таблица.
- Связь с VMT 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*.

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 VMT, из VMT 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*.

- Виртуальные методы незаменимы и 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.
- 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, и 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.

Бесплодные классы

- ключевое слово sealed позволяет описать класс, от которого, в противоположность абстрактному, наследовать запрещается:

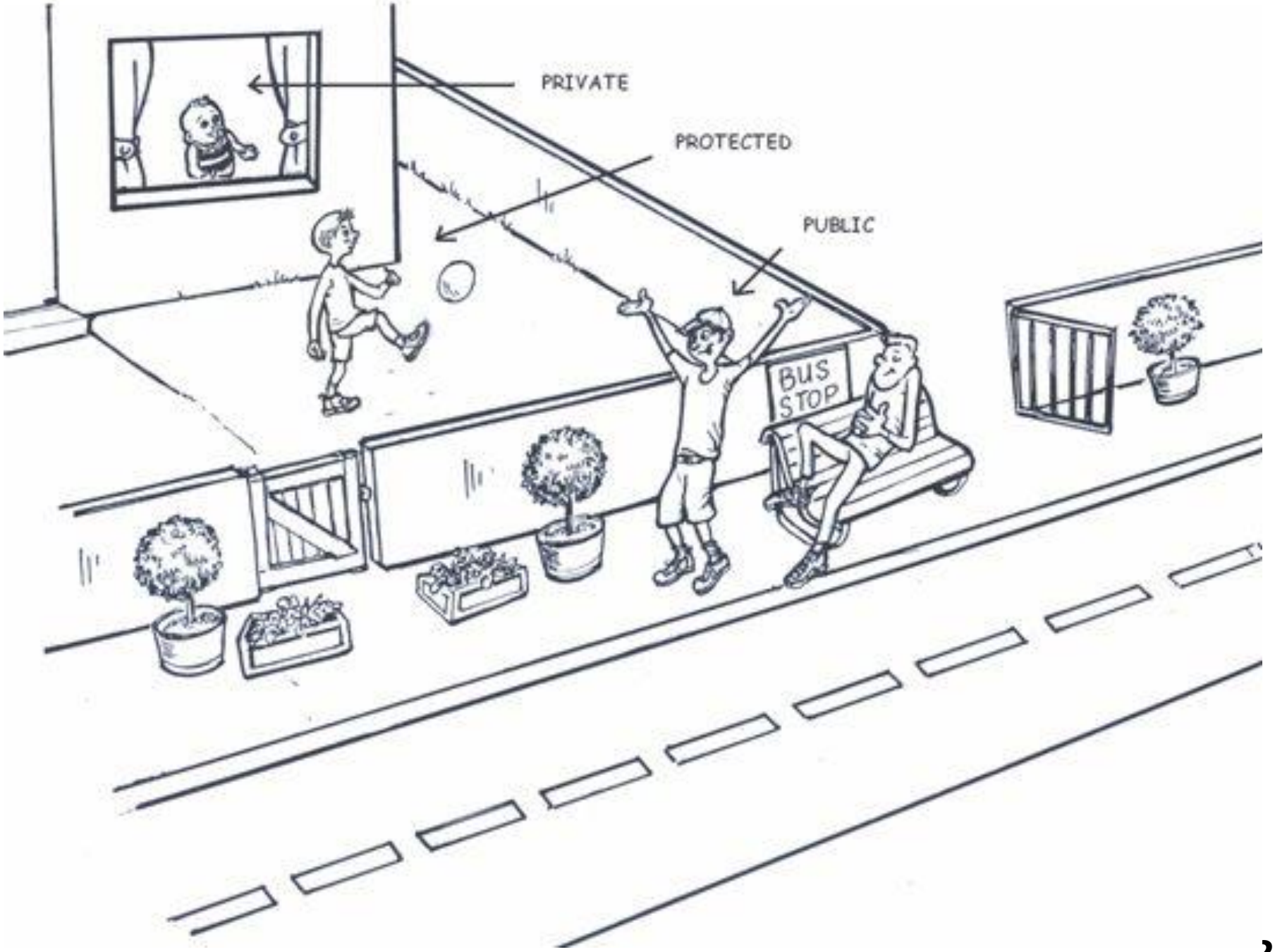
```
sealed class Spirit    {  ...    }

//  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*.

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!!


PRIVATE
PROTECTED
PUBLIC
BUS STOP

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 часто используется и непосредственно при описании типа параметров методов для придания им общности, а также для хранения ссылок на объекты различного типа — таким образом реализуется полиморфизм.

Открытые методы класса System.Object

- Метод Equals с одним параметром возвращает значение true, если параметр и вызывающий объект ссылаются на одну и ту же область памяти:

```
public virtual bool Equals(object obj);
```

- Метод Equals с двумя параметрами возвращает значение true, если оба параметра ссылаются на одну и ту же область памяти:

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

- Метод GetHashCode формирует хэш-код объекта и возвращает число, однозначно идентифицирующее объект:

```
public virtual int GetHashCode();
```

- Метод GetType возвращает текущий полиморфный тип объекта, то есть не тип ссылки, а тип объекта, на который она в данный момент указывает:

```
public Type GetType();
```

- Метод ReferenceEquals возвращает значение true, если оба параметра ссылаются на одну и ту же область памяти:

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

- Метод ToString по умолчанию 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:

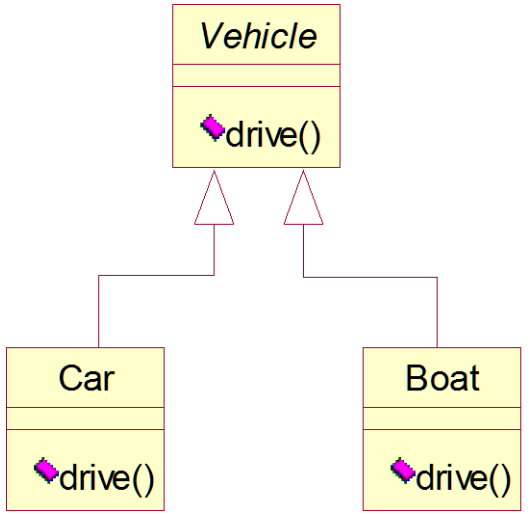
```
public virtual string ToString()
```

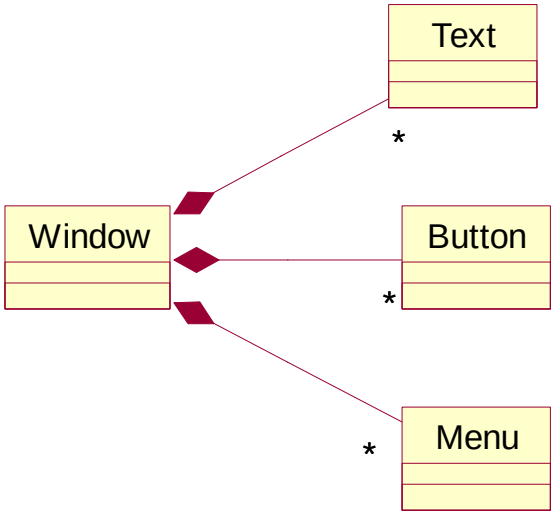
Рекомендации по программированию

- 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 (не виртуальные) методы переопределять в производных классах не рекомендуется.

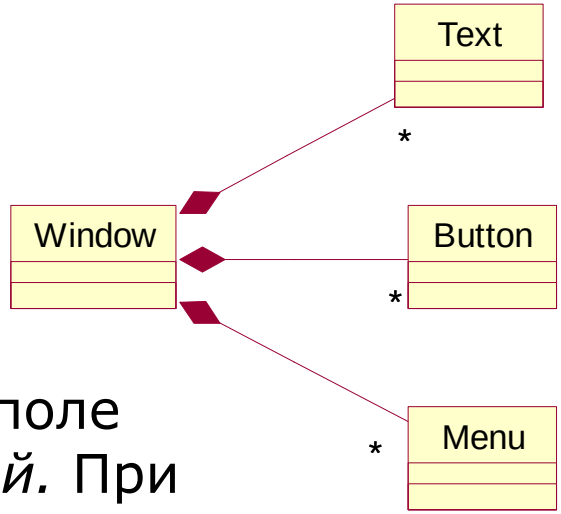
Вложение

- Наследование класса Y от класса X означает, что Y представляет собой разновидность класса X, то есть более конкретную, частную концепцию.
- Альтернативным наследованию механизмом использования одним классом другого является вложение, когда один класс является полем другого.
- *Вложение* представляет отношения классов «Y содержит X» или «Y реализуется посредством X».





Композиция и агрегация



- Вид вложения, когда в классе описано поле объектного типа, называют *композицией.* При композиции время жизни всех объектов — и объемлющего, и его полей — одинаково.
- Если в классе описан указатель на поле объектного типа, это обычно называют *агрегацией*.
- Агрегация представляет собой более слабую связь между объектами. Поле-указатель может также ссылаться не на один объект, а на неопределенное количество объектов, например быть указателем на начало линейного списка.

Интерфейсы


Видеоадаптеры и интерфейсы
Видеоадаптеры и интерфейсы для мониторов

Интерфейсы

- *Интерфейс* является «крайним случаем» абстрактного класса. В нем задается набор абстрактных методов, свойств и индексаторов, которые должны быть реализованы в производных классах.
- интерфейс определяет поведение, которое поддерживается реализующими этот интерфейс классами.
- Основная идея использования интерфейса состоит в том, чтобы к объектам таких классов можно было обращаться одинаковым образом.
- Каждый класс может определять элементы интерфейса по-своему. Так достигается полиморфизм: объекты разных классов по-разному реагируют на вызовы одного и того же метода.
- Синтаксис интерфейса аналогичен синтаксису класса:

[атрибуты] [спецификаторы] interface имя_интерфейса [: **предки]**

тело_интерфейса [; **]**

- Интерфейс может наследовать свойства нескольких интерфейсов, в этом случае *предки* перечисляются через запятую.
- *Тело интерфейса* составляют абстрактные методы, шаблоны свойств и индексаторов, а также события.
- Интерфейс не может содержать константы, поля, операции, конструкторы, деструкторы, типы и любые статические элементы.

```
interface IAction
{
    void Draw();
    int Attack(int a);
    void Die();
    int Power { get; }
}
```

Область применения интерфейсов

- Если некий набор действий имеет смысл только для какой-то конкретной иерархии классов, реализующих эти действия разными способами, уместнее задать этот набор в виде виртуальных методов абстрактного базового класса иерархии.
- То, что работает в пределах иерархии одинаково, предпочтительно полностью определить в базовом классе.
- Интерфейсы же чаще используются для задания общих свойств объектов различных иерархий.

Отличия интерфейса от абстрактного класса

- элементы интерфейса по умолчанию имеют спецификатор доступа public и не могут иметь спецификаторов, заданных явным образом;
- интерфейс не может содержать полей и обычных методов — все элементы интерфейса должны быть абстрактными;
- класс, в списке предков которого задается интерфейс, должен определять *все* его элементы, в то время как потомок абстрактного класса может не переопределять часть абстрактных методов предка (в этом случае производный класс также будет абстрактным);
- класс может иметь в списке предков несколько интерфейсов, при этом он должен определять все их методы.

Реализация интерфейса

- В C# поддерживается одиночное наследование для классов и множественное — для интерфейсов. Это позволяет придать производному классу свойства нескольких базовых интерфейсов, реализуя их по своему усмотрению.
- Сигнатуры методов в интерфейсе и реализации должны полностью совпадать.
- Для реализуемых элементов интерфейса в классе следует указывать спецификатор public.
- К этим элементам можно обращаться как через объект класса, так и через объект типа соответствующего интерфейса.

Пример

```
interface Iaction
  {  void Draw();  int Attack( int a );  void Die();  int Power { get; }  }
   class Monster : IAction
   {
      public void Draw() { Console.WriteLine( "Здесь был " + name ); }
      public int Attack( int ammo_ )  {
         ammo -= ammo_;
         if ( ammo > 0 ) Console.WriteLine( "Ба-бах!" ); else  ammo = 0;
         return ammo;
      }
      public void Die()
           {Console.WriteLine( "Monster " + name + " RIP" ); health = 0;}
      public int Power { get  { return ammo * health; }
   }
```

```
Monster Vasia = new Monster( 50, 50, "Вася" );  // объект класса Monster
      Vasia.Draw();                              // результат: Здесь был Вася
IAction Actor = new Monster( 10, 10, "Маша" );  // объект типа интерфейса
      Actor.Draw();                              // результат: Здесь был Маша
```

Обращение к реализованному методу через объект типа интерфейса

- Удобство этого способа проявляется при присваивании объектам типа IAction ссылок на объекты различных классов, поддерживающих этот интерфейс.
- Например, легко себе представить метод с параметром типа интерфейса. На место этого параметра можно передавать любой объект, реализующий интерфейс:

```
static void Act( IAction A )
{
    A.Draw();
}
static void Main()
{
   Monster Vasia = new Monster( 50, 50, "Вася" );
   Act( Vasia );
   ...
}
```

Второй способ реализации интерфейса

Явное указание имени интерфейса перед реализуемым элементом.

Спецификаторы доступа не указываются. К таким элементам можно обращаться в программе *только через объект типа интерфейса*:

```
class Monster : IAction {
    int IAction.Power { get{ return ammo * health;}}
    void IAction.Draw() {
        Console.WriteLine( "Здесь был " + name );    }          }
...
IAction Actor = new Monster( 10, 10, "Маша" );
Actor.Draw();                    // обращение через объект типа интерфейса
// Monster Vasia = new Monster( 50, 50, "Вася" );
// Vasia.Draw();                                          ошибка!
```

При этом соответствующий метод *не входит в интерфейс класса*. Это позволяет упростить его в том случае, если какие-то элементы интерфейса не требуются конечному пользователю класса.

Кроме того, этот способ позволяет избежать конфликтов при множественном наследовании

Пример

Пусть класс Monster поддерживает два интерфейса: один для управления объектами, а другой для тестирования:

```
interface Itest { void Draw(); }
interface Iaction { void Draw();    int Attack( int a );    void Die();    int Power { get; }}
class Monster : IAction, Itest {
    void ITest.Draw()  {
        Console.WriteLine( "Testing " + name );    }
    void IAction.Draw() {
        Console.WriteLine( "Здесь был " + name );    }
    ... }
```

Оба интерфейса содержат метод Draw с одной и той же сигнатурой. Различать их помогает явное указание имени интерфейса. Обращаться к этим методам можно, используя операцию приведения типа, например:

```
Monster Vasia = new Monster( 50, 50, "Вася" );
((ITest)Vasia).Draw();                    // результат: Здесь был Вася
((IAction)Vasia).Draw();                  // результат:   Testing Вася
```

Операция is

- При работе с объектом через объект типа интерфейса бывает необходимо убедиться, что объект поддерживает данный интерфейс. Проверка выполняется с помощью бинарной операции is. Эта операция определяет, совместим ли текущий тип объекта, находящегося слева от ключевого слова is, с типом, заданным справа.
- Результат операции равен true, если объект можно преобразовать к заданному типу, и false в противном случае. Операция обычно используется в следующем контексте:

```
if ( объект is тип )
{
   // выполнить преобразование "объекта" к "типу"
   // выполнить действия с преобразованным объектом
}
```

Операция as

- Операция as выполняет преобразование к заданному типу, а если это невозможно, формирует результат null, например:

```
static void Act( object A )
{
    IAction Actor = A as IAction;
    if ( Actor != null ) Actor.Draw();
}
```

- Обе рассмотренные операции применяются как к интерфейсам, так и к классам.

Интерфейсы и наследование

- Интерфейс может не иметь или иметь сколько угодно интерфейсов-предков, в последнем случае он наследует все элементы всех своих базовых интерфейсов, начиная с самого верхнего уровня.
- Базовые интерфейсы должны быть доступны в не меньшей степени, чем их потомки.
- Как и в обычной иерархии классов, базовые интерфейсы определяют общее поведение, а их потомки конкретизируют и дополняют его.
- В интерфейсе-потомке можно также указать элементы, переопределяющие унаследованные элементы с такой же сигнатурой. В этом случае перед элементом указывается ключевое слово new, как и в аналогичной ситуации в классах. С помощью этого слова соответствующий элемент базового интерфейса скрывается.

Пример

```
interface IBase  {  void F( int i );  }
interface Ileft : IBase {
   new void F( int i );              /* переопределение метода F */   }
interface Iright : IBase  {   void G();  }
interface Iderived : ILeft, IRight {}
class A  {
   void Test( IDerived d ) {
      d.F( 1 );                    // Вызывается ILeft.F
      ((IBase)d).F( 1 );            // Вызывается IBase.F
      ((ILeft)d).F( 1 );           // Вызывается ILeft.F
      ((IRight)d).F( 1 );           // Вызывается IBase.F
   }
}
```

Метод F из интерфейса IBase скрыт интерфейсом ILeft, несмотря на то, что в цепочке IDerived — IRight — IBase он не переопределялся.

Особенности реализации интерфейсов

- Класс, реализующий интерфейс, должен определять все его элементы, в том числе унаследованные. Если при этом явно указывается имя интерфейса, оно должно ссылаться на тот интерфейс, в котором был описан соответствующий элемент.
- Интерфейс, на собственные или унаследованные элементы которого имеется явная ссылка, должен быть указан в списке предков класса.
- Класс наследует все методы своего предка, в том числе те, которые реализовывали интерфейсы. Он может переопределить эти методы с помощью спецификатора new, но обращаться к ним можно будет только через объект класса.

Стандартные интерфейсы .NET

- В библиотеке классов .NET определено множество стандартных интерфейсов, задающих желаемое поведение объектов. Например, интерфейс IComparable задает метод сравнения объектов по принципу больше или меньше, что позволяет выполнять их сортировку.
- Реализация интерфейсов IEnumerable и IEnumerator дает возможность просматривать содержимое объекта с помощью конструкции foreach, а реализация интерфейса ICloneable — клонировать объекты.
- Стандартные интерфейсы поддерживаются многими стандартными классами библиотеки. Например, работа с массивами с помощью цикла foreach возможна именно потому, что тип Array реализует интерфейсы IEnumerable и IEnumerator.
- Можно создавать и собственные классы, поддерживающие стандартные интерфейсы, что позволит использовать объекты этих классов стандартными способами.

Сравнение объектов

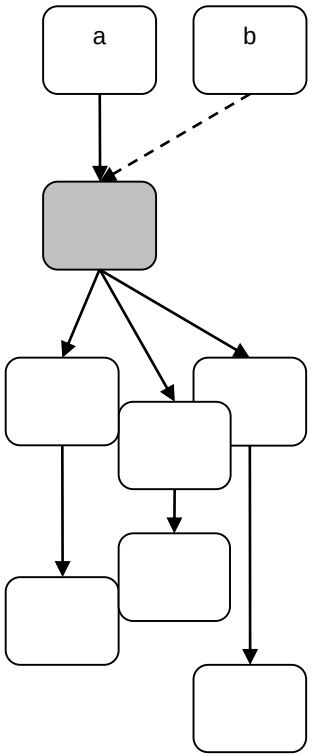
- Интерфейс IComparable определен в пространстве имен System. Он содержит всего один метод CompareTo, возвращающий результат сравнения двух объектов — текущего и переданного ему в качестве параметра:

```
interface IComparable
{
    int CompareTo( object obj )
}
```

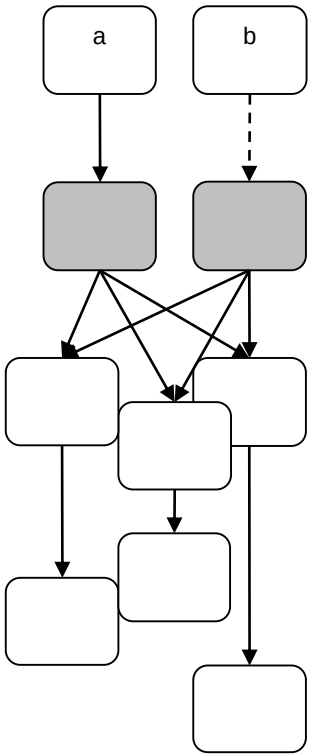
- Метод должен возвращать:
  - 0, если текущий объект и параметр равны;
  - отрицательное число, если текущий объект меньше параметра;
  - положительное число, если текущий объект больше параметра.

Клонирование объектов

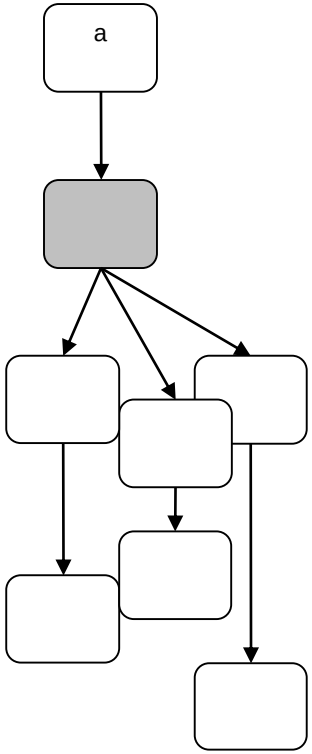
- *Клонирование* — это создание копии объекта. Копия объекта называется клоном.

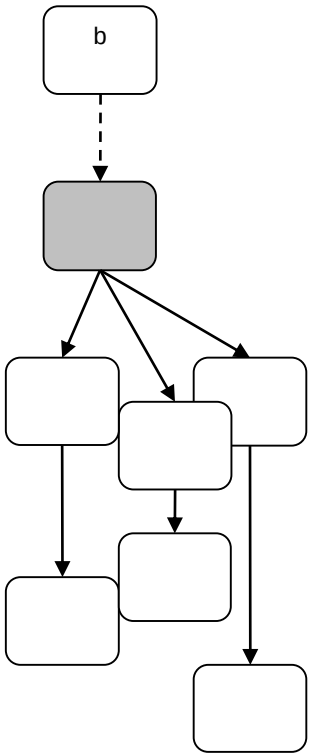


а) присваивание
b = a



б) поверхностное клонирование





в) глубокое клонирование

Виды клонирования

- При присваивании одного объекта ссылочного типа другому копируется ссылка, а не сам объект (рис. *а*).
- Если необходимо скопировать в другую область памяти поля объекта, можно воспользоваться методом MemberwiseClone, который объект наследует от класса object. При этом объекты, на которые указывают поля объекта, в свою очередь являющиеся ссылками, не копируются (рис. *б*). Это называется *поверхностным клонированием*.
- Для создания полностью независимых объектов необходимо *глубокое клонирование*, когда в памяти создается дубликат всего дерева объектов (рис. *в*).
- Алгоритм глубокого клонирования весьма сложен, поскольку требует рекурсивного обхода всех ссылок объекта и отслеживания циклических зависимостей.
- Объект, имеющий собственные алгоритмы клонирования, должен объявляться как наследник интерфейса ICloneable и переопределять его единственный метод Clone.

Делегаты

- *Делегат* — это вид класса, предназначенный для хранения ссылок на методы. Делегат, как и любой другой класс, можно передать в качестве параметра, а затем вызвать инкапсулированный в нем метод. Делегаты используются для поддержки событий, а также как самостоятельная конструкция языка. Рассмотрим сначала второй случай.
- Описание делегата задает сигнатуру методов, которые могут быть вызваны с его помощью:

[атрибуты] [спецификаторы] delegate тип имя_делегата ([параметры])

- Пример описания делегата:
- public delegate void D (int i);
- Базовым классом делегата является класс System.Delegate

Использование делегатов

- Делегаты применяются в основном для следующих целей:
  - получения возможности определять вызываемый метод не при компиляции, а динамически во время выполнения программы;
  - обеспечения связи между объектами по типу «источник — наблюдатель»;
  - создания универсальных методов, в которые можно передавать другие методы;
  - поддержки механизма обратных вызовов.

Операции

- Делегаты можно *сравнивать на равенство и неравенство*. Два делегата равны, если они оба не содержат ссылок на методы или если они содержат ссылки на одни и те же методы в одном и том же порядке.
- С делегатами одного типа можно *выполнять операции* простого и сложного *присваивания*.
- Делегат, как и строка string, является неизменяемым типом данных, поэтому при любом изменении создается новый экземпляр, а старый впоследствии удаляется сборщиком мусора.

События

- *Событие* — это элемент класса, позволяющий ему посылать другим объектам уведомления об изменении своего состояния. При этом для объектов, являющихся наблюдателями события, активизируются методы-обработчики этого события. Обработчики должны быть зарегистрированы в объекте-источнике события. Таким образом, механизм событий формализует на языковом уровне паттерн «наблюдатель».
- Механизм событий можно также описать с помощью модели «публикация — подписка»: один класс, являющийся *отправителем* (sender) сообщения, публикует события, которые он может инициировать, а другие классы, являющиеся *получателями* (receivers) сообщения, подписываются на получение этих событий.

Механизм событий

- События построены на основе делегатов: с помощью делегатов вызываются методы-обработчики событий. Поэтому *создание события* в классе состоит из следующих частей:
  - описание делегата, задающего сигнатуру обработчиков событий;
  - описание события;
  - описание метода (методов), инициирующих событие.
- Синтаксис события:

[атрибуты] [спецификаторы] event тип имя_события

Пример

```
public delegate void Del( object o );     // объявление делегата
class A
{
    public event Del Oops;                 // объявление события
    ...
}
```

- *Обработка событий* выполняется в классах-получателях сообщения. Для этого в них описываются методы-обработчики событий, сигнатура которых соответствует типу делегата. Каждый объект (не класс!), желающий получать сообщение, должен зарегистрировать в объекте-отправителе этот метод.
- Событие — это удобная абстракция для программиста. На самом деле оно состоит из закрытого статического класса, в котором создается экземпляр делегата, и двух методов, предназначенных для добавления и удаления обработчика из списка этого делегата.
- Внешний код может работать с событиями единственным образом: добавлять обработчики в список или удалять их, поскольку вне класса могут использоваться только операции += и -=. Тип результата этих операций — void, в отличие от операций сложного присваивания для арифметических типов. Иного способа доступа к списку обработчиков нет.

Пример

```
public delegate void Del();                    // объявление делегата
   class Subj                                   // класс-источник
   {
      public event Del Oops;                    // объявление события
      public void CryOops()                  // метод, инициирующий событие
      { Console.WriteLine( "OOPS!" ); if ( Oops != null ) Oops();      }
   }
   class ObsA                                   // класс-наблюдатель
   {
      public void Do();                      // реакция на событие источника
      {  Console.WriteLine( "Вижу, что OOPS!" );  }
   }
   class ObsB                                   // класс-наблюдатель
   {
      public static void See()             // реакция на событие источника
      {  Console.WriteLine( "Я тоже вижу, что OOPS!" );   }
   }
```

```
class Class1
   {
      static void Main()
      {
         Subj s  = new Subj();              //    объект класса-источника
         ObsA o1 = new ObsA();               //                  объекты
         ObsA o2 = new ObsA();               //        класса-наблюдателя
         s.Oops += new Del( o1.Do );         //                добавление
         s.Oops += new Del( o2.Do );         //              обработчиков
         s.Oops += new Del( ObsB.See );      //                  к событию
         s.CryOops();                        //      инициирование события
      }
   }
```