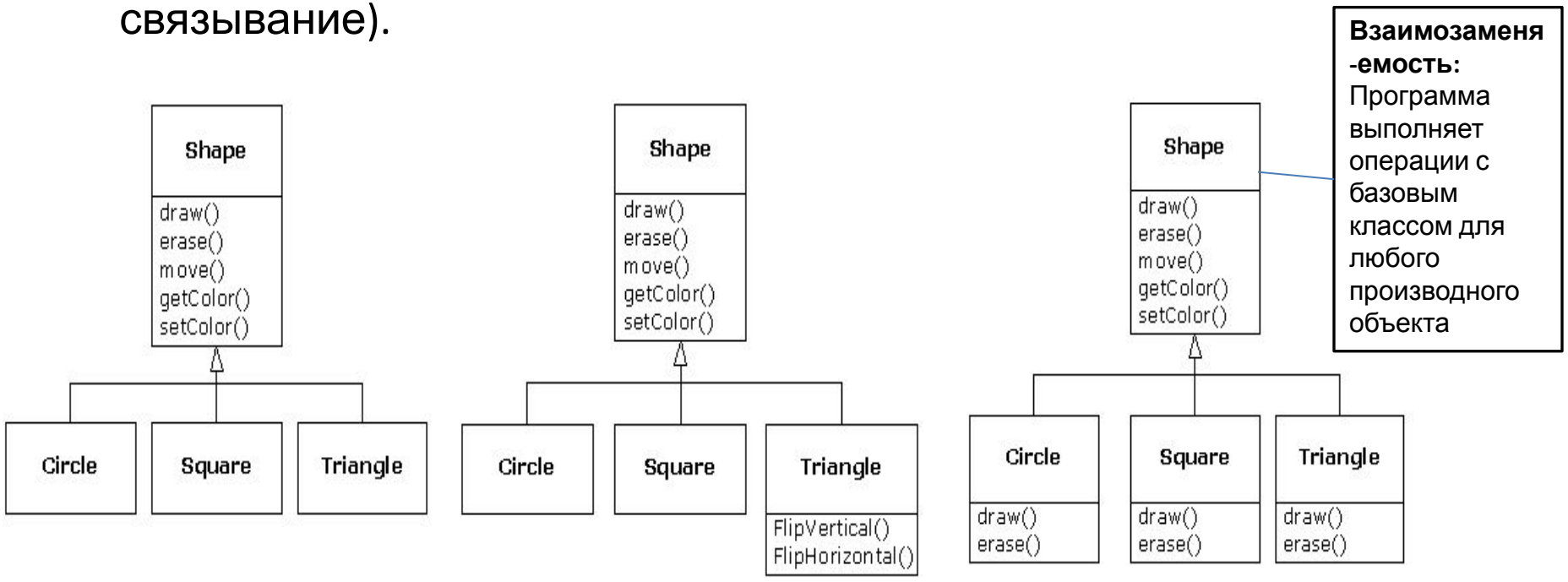
Наследование классов

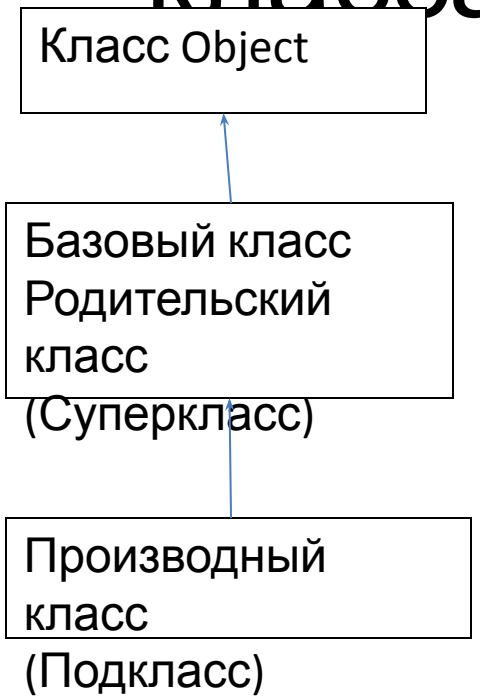
Механизм наследования поддерживает концепцию иерархии классов.

Наследование используются в следующих случаях:

1. Расширение базового класса путем добавления новых данных и методов (повторное использование кода).
2. Изменение поведения функций базового класса (перегрузка функций).
3. Взаимозаменяемые объекты (полиморфизм и позднее связывание).



Спецификация базового класса



Суперкласс указывается ключевым словом extends. Он должен быть доступным классом и не иметь модификатора final.

Классы, для которых не указан расширяемый класс, являются неявным расширением класса Object.

```
Object oref = new A();
```

Доступ к компонентам базового класса

super — используется как ссылка на экземпляр суперкласса с целью обеспечения доступа к одноименным нестатическим полям и методам суперкласса.

```
class ClassA {
 float x;
 ...
}
class ClassB extends ClassA {
 int x;
 ...
 public void method l(int x) {
 int iX1 = x;    // присваивание значения параметра метода
 int iX2 = this.x;  // присваивание значения поля данного класса
 float fX = super.x; // присваивание значения поля суперкласса
 }
}
class ClassA {
 ...
 public void method2() {...} ...
}
class ClassB extends ClassA {
 ...
 public void method2() {
  super.method2(); // вызов метода суперкласса
 }
}
```

Конструкторы суперкласса

В производном классе конструктор суперкласса, отличный от конструктора по умолчанию, должен быть вызван явно с помощью метода **super()**.

```
public class Person {
private String name;
public Person(String name) {
this.name = name;
} }
public class Employee extends Person {
public Employee(String name) {
super(name);
} }
```

Суперкласс с внутренним классом

```
import java.awt.Color;
public class Rect {
class MyRect   //  Внутренний класс Бесцветный прямоугольник
{
  protected int x1,y1,x2,y2;
  public String str = "Бесцветный";
MyRect(int x1, int y1, int x2, int y2)
{
  this.x1 = x1;   this.y1 = y1;   this.x2 = x2;   this.y2 = y2;
}
  public void setMyRect(int vx1, int vy1, int vx2, int vy2)
{
  x1 = vx1;   y1 = vy1;   x2 = vx2;   y2 = vy2;
}
  public String toString()
{
  String sz = "Прямоугольник: (" +  x1 + ", " + y1 + ", " + x2 + ", " + y2 + ")";
       return sz;
}
}
```

Описание подкласса MyColorRect

```
class MyColorRect extends MyRect { // описывает цветные прямоугольники
  protected Color rectColor=Color.white;
  public String str = "Цветной";
   MyColorRect(int x1, int y1,int x2, int y2, Color colr)
  {   super(x1, y1, x2, y2);
  rectColor = colr;   }
  MyColorRect()
  {   super(0, 0, 0, 0);   }
  public void setColor(Color colr)
  {   rectColor = colr;
  }
   public String toString() // переопределяет toString() в классе MyRect
  {   String sz = super.toString()+ rectColor.toString();
     return sz;
  }
}
```

Создание объектов базовых и производных классов

```
public static void main(String[] args) {
    MyColorRect rect1 = new Rect().new MyColorRect(0, 0, 10, 20, Color.black);
      String szStr = rect1.toString();
      System.out.println(szStr);
      MyColorRect  rect2 =new Rect(). new MyColorRect();
      String szStr2 = rect2.toString();
      System.out.println(szStr2);
      MyRect rect;
      MyRect rect3=new Rect().new MyRect(1,1,2,2);
      rect=rect3;
      String szStr4 = rect.toString();
      System.out.println(szStr4+rect.str+ rect.x1);
      rect=rect1;
      String szStr3 = rect.toString(); // вызывается из MyColorRect
      System.out.println(szStr3+ rect.str + rect.x1);
      }
    }
```

Абстрактные классы

Если метод не имеет тело, то его нужно пометить как abstract (класс, который содержит данный метод, должен быть тоже объявлен как абстрактный).
В производном классе этот метод должен быть переопределен.

```
abstract class Fruits
{
abstract void setSize (String size);
public void Message ()
{
System.out.println ("Это фрукт!");
}
}
class Lemon extends Fruits
{
public String size;
public void setSize (String size)
{
this.size = "Этот лимон такого размера: " + size;
}
public static void main (String args[])
{
Lemon myLemon = new Lemon ();
myLemon.setSize ("большой");
System.out.println (myLemon.size);
}
}
```

Приведение типов

При приведении ссылочных типов действуют следующие правила:

- объект всегда может быть приведен к типу своего непосредственного суперкласса;
- приведение ссылочных типов может выполняться по иерархии наследования классов сколь угодно глубоко;
- любой класс ссылочного типа всегда можно привести к типу Object.

1. После приведения объекта к типу суперкласса все переменные и методы самого класса объекта становятся недоступными для приведенного объекта.
2. Значение простого типа не может быть присвоено значению ссылочного типа, как и значение ссылочного типа не может быть присвоено значению простого типа.

Оператор instanceof

С помощью оператора instanceof можно определить

- ✔ Принадлежит объект указанному типу
- ✔ Является ли объект подклассом указанного класса
- ✔ Реализует ли объект интерфейс

Имя объекта **instanceof** *Имя ссылочно типа*

```
String str1;
Object ObjectName;
ObjectName =(Object) str1; // Приведение типа
if (ObjectName instanceof String)  // Проверка на совместимость String
String str2 = (String) ObjectName ;
```

Пример использования оператора instanceof

```
class A {  int i, j; }
class B {   int i, j; }
class C extends A {   int k; }
class D extends A {   int k; }
class InstanceOf {
  public static void main(String args[]) {
    A a = new A();     B b = new B();     C c = new C();     D d = new D();
    if (a instanceof A)       System.out.println("a экземпляр A");
if (c instanceof A)       System.out.println("c  может приведен к  A");
    if (a instanceof C)       System.out.println("a может приведен к C");

    A ob;
    ob = d; // ob ссылается на A часть объекта d
       if (ob instanceof D)       System.out.println("ob совмести с экземпляром D");
     ob = c; // ob ссылается на A часть объекта c
       if (ob instanceof D)       System.out.println("ob совмести с экземпляром  D");
    else       System.out.println("ob не совместим с экземпляром D");
    if (ob instanceof A)       System.out.println("ob совместим  экзпляром A");

    // все объекты могут быть приведеныObject
    if (a instanceof Object)       System.out.println(" a сместим Object");
} }
```

Интерфейсы

[модификатор] **interface** ИмяНовогоИнтерфейса

[**extends** список Интерфейсов]

{Тело интерфейса, состоящее из описаний абстрактных методов и констант}

Интерфейс позволяет иметь различные реализации методов в разных классах и обращаться через него к объекту.

Интерфейсы имеют следующие ограничения:

- Модификатор доступа — могут быть только public или отсутствовать (тогда, поумолчанию, интерфейс доступен только членам пакета, в котором он объявлен).
- Методы — могут быть только абстрактными методами;
- Поля — final, static (константы, не меняющие значений, такие спецификации для них назначаются автоматически, должны быть инициализированы постоянными значениями);
- Сами интерфейсы — не могут иметь конструкторы и реализации методов.
- Нельзя создать объект типа интерфейса (но можно использовать в качестве типа — интерфейсные ссылки).

Интерфейсные константы

Интерфейсы можно использовать для импорта в различные классы совместно используемых констант.

```
public interface MyConstants
        {
   public static final double price = 1450.00;
   public static final int counter = 5;
        }
interface MyColors {
   int RED = 1, YELLOW = 2, BLUE = 4;
}
```

Описание и реализация методов интерфейса

```
public interface MyInterface
        {
    abstract public void add(int x, int y);
             void volume(int x,int y, int z);
        }
class Demo1 implements MyInterface                  class Demo2 implements MyInterface
    {                                  {
    public void add(int x, int y)                  public void add(int x, int y)
        {                                  {
    System.out.println(     +(x+y));               System.out.println(     +(x*y));
        }                                  }

    public void volume(int x, int y, int z)            public void volume(int x, int y, int z)
        {                                  {
    System.out.println(     +(x*y*z));             System.out.println(     +(x-y-z));
        }                                  }

public static void main(String args[])
        {
     MyInterface  d1= new Demo1();                  MyInterface d2= new Demo2();
    d1.add(10,20);                          d2.add(10,20);
    d1.volume(10,10,10);                        d2.volume(10,10,10);
        } }
```

- Интерфейс можно использовать как ссылочный тип при объявлении переменных.
- Переменная или выражение типа интерфейса могут ссылаться на любой объект, который является экземпляром класса, реализующего данный интерфейс.
- Переменную типа интерфейса можно использовать только после присвоения ей ссылки на объект ссылочного типа, для которого был реализован данный интерфейс.

Вложенные интерфейсы

Можно вкладывать описание интерфейса внутрь описания класса или другого интерфейса.

```
/описание класса
class SomeClass {
void MethodSomeClass(){}
 //описание вложенного интерфейса
interface SomeClassItf{
void SomeMethod();
} }

 //описание внешнего интерфейса
interface OuterInterface {
void OuterInterfaceMethod();
 //описание вложенного интерфейса
interface InnerInterface {
void InnerInterfaceMethod();
} }
class A implements OuterInterface.InnerInterface, SomeClass.SomeClassItf {
… // реализация InnerInterfaceMethod и SomeMethod
}
```

Использование вложенного интерфейса идет через имя внешнего класса или интерфейса:

```
SomeClass.SomeClassItf si = new A();
si.SomeMethod();

OuterInterface.InnerInterface ii = new A();
ii.InnerInterfaceMethod();
```

Наследование интерфейсов

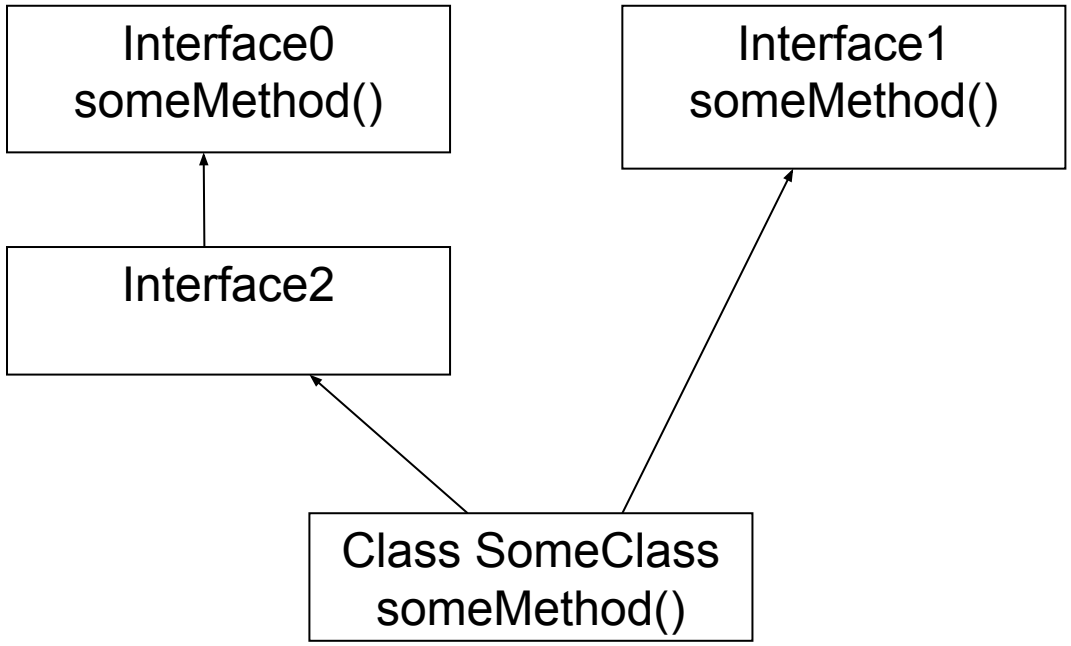
```
//суперинтерфейс A
interface A {
 int a_value = 1;
 void A();
}
 //интерфейс B расширяет интерфейс A
interface B extends A{
 int b_value = 2;
 void B();
}
 //интерфейс C расширяет интерфейс B
interface C extends B{
 int c_value = 3;
 void C();
}

class Test implements C{ . . . }
Test t = new Test();
 t.A();
 t.B();
 t.C();
```

Использование констант при множественном наследовании интерфейсов

```
public interface I1 {
   Double PI=3.14;
}
public interface I2 {
   Double PI=3.1415;
}
class C1 implements I1,I2 {
  void m1(){
    System.out.println("I1.PI="+ I1.PI);
    System.out.println("I2.PI="+ I2.PI);
  };
}
```

Для использования констант с одинаковыми именами из разных интерфейсов решением является квалификация имени константы именем соответствующего интерфейса

Наследование интерфейсов и реализация интерфейсов

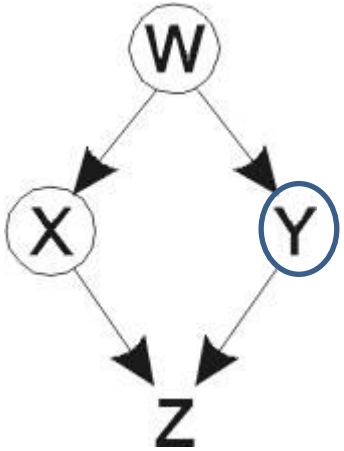


- Класс должен полностью реализовать все методы интерфейса, либо часть методов, но в этом случае должен быть объявлен как абстрактный.
- Если класс реализует несколько интерфейсов, в которых есть одноимённые методы, то в нём может задаваться лишь одна реализация общая для всех этих методов

Использование переменных типа интерфейс

```
interface Interface0 {    int someField = 10;     String someMethod(); }
interface Interface1 {     int someField = 100;     String someMethod(); }
interface Interface2 extends Interface0{     int someField = 200;     String someMethod(); }
class SomeClass implements Interface1, Interface2 {
   public String someMethod() {    return "Метод";     }
}
public class Main {
      public static void main(String[] args) {
        SomeClass a = new SomeClass();
       Interface1 I1=a;
         System.out.println( a.someMethod() );                              // Метод
    //   System.out.println( a.someField );                                 // ошибка
      System.out.println( ( (Interface1) a).someField );                  // 100
       System.out.println( Interface1.someField );                         // 100
      Interface2 I2=a;
      System.out.println( I2.someField );                  // 200
       System.out.println( I2.someMethod() );                    // Метод
       Interface0 I0=a;
       System.out.println( I0.someField );                  // 10
       System.out.println( Interface0.someField );               // 10
      }
}
```

Конфликты имен




```
interface W { }
interface X extends W { }
interface Y extends W { }
class Z implements X, Y { }
```

Если интерфейсы X и Y содержат одноименные методы с разным количеством или типом параметров, то Z будет содержать два перегруженных метода с одинаковыми именами, но разными сигнатурами.
Если же сигнатуры в точности совпадают, то Z может содержать лишь один метод с данной сигнатурой.
Если методы отличаются лишь типом возвращаемого значения, вы не можете реализовать оба интерфейса.

Если два метода отличаются только типом возбуждаемых исключений, метод класса обязан соответствовать обоим объявлениям с одинаковыми сигнатурами , но может иметь не больший список возможных исключений.

Пример конфликтов имен

```
interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C { public int f() { return 1; } }
class C2 implements I1, I2 {
public void f() {}
public int f(int i) { return 1; } // перегружен }
class C3 extends C implements I2 {
public int f(int i) { return 1; } // перегружен }
class C4 extends C implements I3 {
// Одинаковы, нет проблем:
public int f() { return 1; } }
 // Методы различаются только возвращаемым типом:
class C5 extends C implements I1 {} //Ошибка
interface I4 extends I1, I3 {} //Ошибка
```