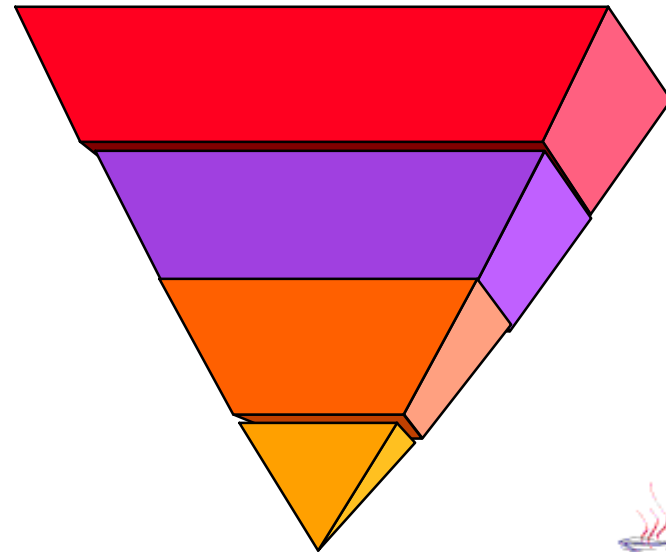


继承 与 多态



- 继承
 - 派生子类
 - 域的继承与隐藏
 - 方法的继承与覆盖
 - this 与 supper
- 多态
- 重载与覆盖
- 类的设计(Isa/HasA, access control)
- 包
- 接口



- 扩展类/超类
 - 父类 / 子类
 - 单继承/多继承
 - 继承层次结构(tree-like/network-like)
- 关系
 - 子类对象 *是* 一个父类对象
 - 一个父类对象 *不是* 一个子类对象
 - 使用显式转换可以将一个父类引用转换为一个子类的引用



PhoneCard

电话卡

剩余金额

拨打电话

查询余额

[返回](#)

继承

None Number PhoneCard

无卡号电话卡

对应电话机型号

获得电话机型号

继承

Number PhoneCard

有卡号电话卡

卡号、密码、接
入号码、接通

登录交换机

[返回](#)

继承

继承

继承

继承

电话磁卡

使用地域

拨打电话

电话卡

拨打电话

magCard

IC Card

IP卡

失效日期

拨打电话

IP Card

200卡

附加费

拨打电话

修改密码

D200Card

[返回](#)

```
abstract class PhoneCard
{
    double balance;
    abstract boolean performDial();
    double getBalance()
    {
        return balance;
    }
}
```

```
abstract class None_Number_PhoneCard extends PhoneCard
{
    String phoneSetType;
    String getSetType()
    {
        return phoneSetType;
    }
}
```



```
abstract class Number_PhoneCard extends PhoneCard
{
    long cardNumber;
    int password;
    String connectNumber;
    boolean connected;
    boolean performConnection(long cn,int pw)
    { if (cn == cardNumber && pw == password)
        {
            connected = true;
            return true;    }
        else
            return false;  }
}
```

[返回](#)

```
class magCard extends None_Number_PhoneCard
{
    double balance;
    String usefulArea;
    boolean performDial()
    {
        if (balance > 0.9)
        {
            balance -= 0.9;
            return true;
        }
        else
            return false;
    }
}
```

[返回](#)

```
class IC_Card extends None_Number_PhoneCard
{
    boolean performDial()
    {
        if (balance > 0.5)
        {
            balance -= 0.9;
            return true;
        }
        else
            return false;
    }
}
```

[返回](#)


```
class IP_Card extends Number_PhoneCard
{
    Date expireDate;
    boolean performDial()
    {
        if (balance > 0.3 && expireDate.after(new Date()))
        {
            balance -= 0.3;
            return true;
        }
        else
            return false;
    }
}
```

[返回](#)

```
class D200_Card extends Number_PhoneCard
{
    double balance; //Number_PhoneCard
    double balance;
    double additoryFee;
    boolean performDial()
    {
        if (balance > (0.5 + additoryFee))
        {
            balance -= (0.5 + additoryFee);
            return true;
        }
        else
            reurn false;
    }
}
```

[返回](#)

- 域的继承: magCard

- 子类可以继承父类的所有非私有域

- 域的隐藏

- 子类重新定义一个与从父类那里继承来的域变量完全相同的变量，称为域的隐藏

- 注意

- 被隐藏的父类的域仍然存在

- 被隐藏的域可以通过super或父类的对象来引用

- 当对象的一个域被访问，对象的声明类型决定是访问父类的域还是子类的域



- P93 例5-2

Public class TestHiddenField

```
{ public static void main(String args[ ] )
```

```
{ D200_Card my200 = new D200_Card();
```

```
    my200.balance = 50.0;
```

```
    System.out.println("Supper:" + my200.getBalance());
```

0.0

```
    if (my200.performDial())
```

```
        System.out.println("Subclass:" + my200.balance);
```

49.5

```
    }
```

```
}
```



- 方法的继承：[IC-Card](#)
- 方法的覆盖
 - 子类重新定义与父类同名的方法
 - 要求
 - 新旧方法必须有相同方法名、参数表、返回类型 (如果只有方法名相同，则是重载)
 - 方法必须是 *non-static*



```
class SuperShow {  
    public String str = "SuperStr";  
    public void show() {  
        System.out.println("Super.show:" + str);  
    }  
}
```

```
class ExtendShow extends SuperShow {  
    public String str = "ExtendStr";  
    public void show() {  
        System.out.println("Extend.show" + str);  
    }  
    public static void main(String[] args) {  
        ExtendShow ext = new ExtendShow();  
        SuperShow sup = ext;  
    }  
}
```



```
        sup.show();//show of actual type, i.e. ExtendShow
        ext.show();
        System.out.println("sup.str="+sup.str);
        System.out.println("ext.str =" +ext.str);
    }
}
```

- Two kinds of reference: actual type(ext), super type(sup)
- Result: **Extend.show:ExtendStr** *// actual type: Extend*

Extend.show:ExtendStr

sup.str = SuperStr *// declared type: SuperShow*

ext.str = ExtendStr *// declared type: ExtendShow*



- **this**代表当前对象
 - 如果使用本类的变量和方法，在其前面隐含着**this**

```
class X {  
    int x;  
    void show() {...}  
    void method() {  
        x = 3;        //相当于this.x = 3;  
        show();      //相当于this.show();  
    }  
}
```



- **this**是一个对象的若干个引用之一，利用**this**可以调用当前对象的方法或使用当前对象的域

- **Example**

```
class Moose {  
    String hairdresser ;  
    Moose(String hairdresser) {  
        this.hairdresser = hairdresser;  
    }  
}
```

field

parameter



- **super**

- 表示当前对象的直接父类对象，是当前对象的直接父类对象的引用

- **使用**

- 用来访问父类被隐藏的成员变量

super.variable

- 用来调用父类中被重写的方法

super.method ([paramlist]) ;

- 用来调用父类的构造方法

super ([paramlist]) ;



```
class D200_Card extends Number_PhoneCard
{
    double balance;
    double additoryFee;
    boolean performDial()
    {
        if (balance > (0.5 + additoryFee))
        {
            balance -= (0.5 + additoryFee);
            return true;
        }
        else
            reurn false;
    }
    double getBalance() {
        return super.balance;
    }
}
```



- 同一个类中存在着 多个具有不同参数列表的构造函数

```
D200_Card() { } //没有形参的构造函数
```

```
D200_Card(long cn) { //一个参数的构造函数
```

```
    this();
```

```
    cardNumber = cn;    }
```

```
D200_Card(long cn, int pw) { //两个参数的构造函数
```

```
    this(cn);
```

```
    password = pw;    }
```

```
D200_Card(long cn, int pw, double b, String c) { //四个参数
```

```
    this(cn, pw, b);
```

```
    connectNumber = c; }
```



- 不专门定义自己的构造函数，自动调用父类的无参构造函数
- 定义自己的构造函数并调用父类有参数构造函数
 - 使用`super`调用父类有参数构造函数，该语句必须是该子类构造函数的第一条语句
 - 可以使用`this`调用子类中重载的其他构造函数

```
D200_Card(long cn, int pass, double b, double a) {  
    super(cn, pass,b); //Number_PhoneCard(long cn, int pw, double b)  
    additoryFee = a;  
}
```



- 当创建一个对象时，对象的各个域根据其类型被设置成相应的缺省初始值(0,\u0000,false,null), 然后调用构造函数。每个构造函数有三个执行阶段：
- 调用父类的构造函数
- 由初始化语句对各域进行初始化
- 执行构造函数的程序体



```
class X {  
    protected int xMask= 0x00ff;  
    protected int fullMask;  
    public X() {  
        fullMask = xMask;    }  
    public int mask(int orig) {  
        return (orig & fullMask);  
    }  
}
```

```
class Y extends X {  
    protected int yMask= 0xff00;  
    public Y() {  
        fullMask |= yMask;  
    }  
}
```

? How is a Y object to be created?



Step	Action	xMask	yMask	fullMask
0	Default	0	0	0
1	Call Y constructor	0	0	0
2	Call X constructor	0	0	0
3	Initialize X fields	0x00ff	0	0
4	Execute X Constructor	0x00ff	0	0x00ff
5	Initialize Y fields	0x00ff	0xff00	0x00ff
6	Execute Y constructor	0x00ff	0xff00	0xffff



- 对象引用类型的类型转换与简单类型一样，包括自动类型转换和强制类型转换（造型）
- 自动类型转换
 - 将一个子类对象分配给一个父类对象
 - 安全转换
- 造型
 - 将一个父类对象分配给一个子类变量
 - 不安全转换
 - 例：`Employee john;`
`class Manager extends Employee,...`
`Manager boss = (Manager) john;`
 - 使用instance of 运算符



```
Class A {  
    public void f() { };  
    static void g(A i) {  
        i.f(); }  
}
```

```
Class B extends A {  
    public static void main(String[] args) {  
        B b = new B();  
        A.g(b);  
    }  
}
```

[返回](#)

- 运算符

- `i instanceof C`

- `i` 是类 `C` 的一个实例

- 例如

- `if (john instanceof Manager) {`

- `Manager boss = (Manager) john; }`

- 注意

- *`null instanceof Type`* 是错误的



- 多态

- 编译时多态（静态多态性）

- 运行时多态（动态多态性）

- 运行时多态是一种在运行时而不是在编译时调用重写方法的一种机制

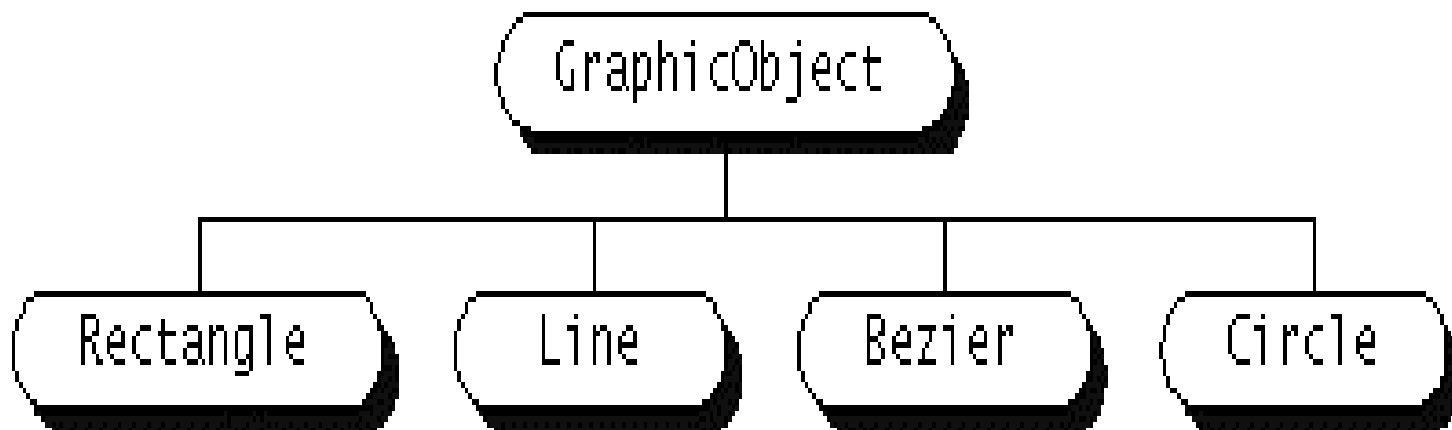
- 子类对象可以当作父类对象使用

- 父类对象不能被当作是其某一个子类的对象

- 如果一个方法的形式参数定义的是父类对象，则调用这个方法时，可使用子类对象作为实际参数

- 如果父类对象引用指向的实际是一个子类对象，则这个父类对象的引用可以用强制类型转换为子类对象的引用





```
GraphicObject g = new Circle();  
g.draw();
```

Casting Up

Circle.draw()

在C++中，用虚函数实现动态绑定；

在JAVA中，缺省情况就是动态绑定；否则可用final定义



- **IsA and HasA**

- (1) 不同的含义

- 继承 vs. 组成
- Example: Point, Pixel(ColorPoint), Circle

- (2) 设计

- **IsA: Employee--Manager, Engineer, Clerk,...**

IsA Fact: an engineer is an employee

IsA Problem: An engineer may be a manager as well

- **HasA:**

Role-Employee(contains several Role Objects)

HasA Fact: an employee may play several roles



```
class Employee {
    String name;
    String dept; ...}
class Manager extends Employee {
    int level;
    String secretaryname;
    Day workday; ... }
class Engineer extends Employee {
    String speciality;
    Day workday; ... }
```

```
class Role {
    String Rolename;
    Day workday; ...}
class Manager extends Role {
    int level;
    String secretaryname; ... }
class Engineer extends Role {
    String speciality;
    ... }
Class Employee {
    String name;
    String dept;
    Role business;
    Role posttitle;...}
```



• 什么是接口

- 一个接口是一个类型, 由抽象方法和常量组成

• 声明

[public] interface <name> [extends <superinterface name>]
{ [<return-type><method-name>(<parameter-list>);]* }

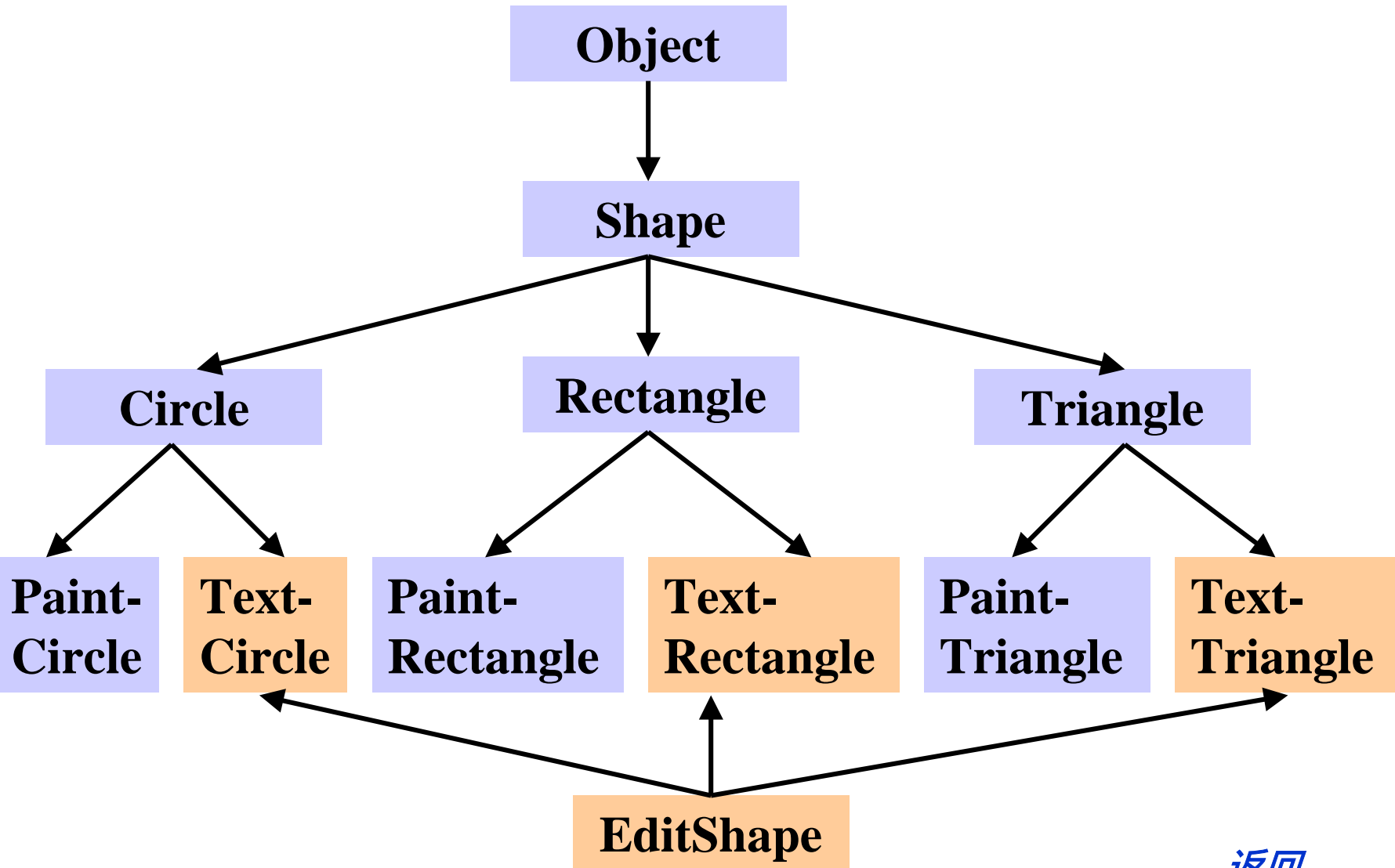
[<type> <final-var> = <value>;]* } // []* -- 0 or many times

Methods: abstract, public, native

Variables: public, static, final (constants)

- 在接口里没有构造函数
- 接口类型可以用来声明变量



[返回](#)

```
interface Attributed {
    void add(Attr newAttr);
    // add an attribute
    Attr find(String attrName);
    // find an attribute by name
    Attr remove(String
                  attrName);
    // find an attribute by name
    java.util.Enumeration attrs();
    // return the current attrs }
    // java.util.Enumeration is also an
interface
```

```
interface Verbose {
    int SILENT = 0;
    int TERSE=1;
    int NORMAL=2;
    int VERBOSE=3;
    void setVerbosity(int
                      level);
    int getVerbosity();
}
```



接口的功能主要体现在以下几个方面：

- 通过接口可以实现不相关类的相同行为，而不需要考虑这些类之间的层次关系
- 通过接口可以指明多个类需要实现的方法
- 通过接口可以了解对象的交互界面，而不需要了解对象所对应的类



- 接口只有被一个类实现后才有意义

```
class <classname> [extends <superclass>]  
    [implements <interface-list>] {    <class_body> }
```

```
interface Callback {  
    void callback(int param);  
}
```

```
class Client implements Callback {  
    public void callback(int p){  
        System.out.println("called with"+p); }  
}
```

...

```
Callback c = new Client();
```

```
c.callback(42); // c can't access other methods of Client
```



- 实现接口的方法是 *public*
- 方法(实现接口) 必须与接口中的抽象方法具有相同的 *signature*
- 一个类可以实现多个接口
- 一个接口可以被多个类实现
- 在实现接口的类中接口变量通常是 *constants* (如 *#define* in C)
- 如果一个类实现一个接口但没有实现所有的接口的方法, 这个类是一个抽象类
- 如果一个类实现几个接口, 每个接口都有同一个方法声明, 则该方法的实现可以被这些接口的所有对象使用



```
import java.util.Random;
interface S {
    int NO = 0;
    int YES = 1;
    int NEVER = 3; }
```

```
class Q implements S {
    Random rand = new Random();
    int ask() {
        int prob=(int) (100*rand.nextDouble());
        if (prob < 30) return NO;
        else if (prob<70) return YES;
        else return NEVER;}
}
```

```
Class A implements S {
    static void a(int result) {
        switch(result) {
            case NO: ....;
                break;
            case YES: ...;
                break;
            case NEVER: ...;
                break; }
        }
    ...
    Q q = new Q();
    a(q.ask());
    a(q.ask());
    ...}
```



(1) 多继承

- 不止一个父类
- 冲突: $X(m), Y(m) \rightarrow Z(m?)$
- 解决方法: 继承约定而不是实现

(2) 接口的继承

```
interface <name> extends <interface-name-list> {  
    [<return-type><method-name>(<parameter-list>); ]*  
    [<type> <final-var> = <value>; ]*  
}
```



```
interface Monster {  
    void menace(); }  
  
interface Dmonster extends Monster{  
    void destory(); }  
  
interface Lethal{  
    void kill(); }  
  
class Dzilla implements Dmonster {  
    public void menace() { };  
    public void destory() { }; }  
  
interface Vampire extends Dmonster,Lethal {  
    void drinkBlood(); }
```



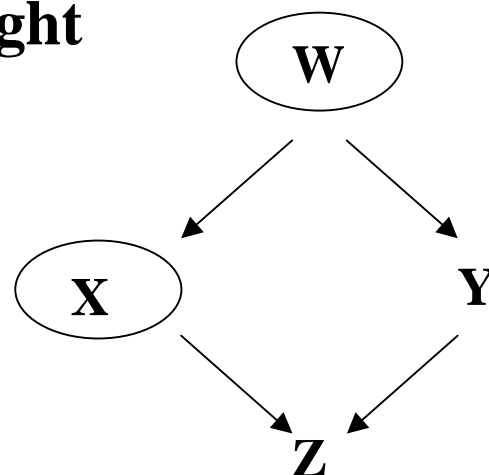
- 多继承: 一个接口可以扩展多个接口
- 新的接口是由它所继承的接口中的所有抽象方法和常量以及它自己定义的方法和常量组成
- 虽然没有root 接口, 一个接口类型的表达式仍可以传给一个具有Object类型参数的方法

```
protected void twiddle(W wRef) { // right  
    Object obj = wRef; // ... }
```

• Example :

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

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



- 一个接口(Z) 可能是两个接口X,Y的子类型. 如果在两个接口(X,Y) 中都包含一个名为M的方法和 一个名为C的常量, 那么在Z中会产生什么问题?
- 如果X中的M的 *signature* 与Y中的M相同, Z 中将有一个此*signature* 的方法
- 如果X中的M的参数个数或类型与Y中的M不同, Z 中将有两个名字相同的重载方法
- 如果X中的M与Y中的M的返回类型不相同, 那么这两个接口就不能同时实现
- 如果X中的M与Y中的M引发的异常类型不同, 则类必须有一种唯一的实现来同时满足这两种引发的异常类型



```
interface X {  
    void setup() throws SomeException; }  
  
interface Y {  
    void setup(); }  
  
class Z implements X,Y {  
    public void setup() { //meets both X.setup and Y.setup  
        //... it can have fewer exception types than supers  
    } // impossible if X.setup and Y.setup have different  
    } // meanings or incompatible exception types
```

- 如果 X和Y 中有相同的常量C , 但是X.C != Y.C , 则Z可以同时实现X和Y, 并且可以在Z中通过X.C and Y.C来访问 C



- 多继承 vs. 单继承
- 抽象类 可以包含方法的实现, protected 成员, static 方法, 而接口 中只有 public 方法和常量
- 它们都是Java 中的重要组成部分
- 一个变量可以被声明为接口类型来引用一个实现该接口的任何类的对象。这样不同类中的相同方法可以被调用。(接口被设计成支持动态方法调用)



```
public abstract class Shape {  
    public double area() {return 0.0;}  
    public double volume() {return 0.0; }  
    public abstract String getName(); }
```

```
public class Point extends Shape {  
    protected double x,y //coordinates of the circle  
    public Point(double a, double b) {setPoint(a,b);}  
    public void setPoint (double a, double b) {  
        x = a; y = b; }  
    public double getX() {return x;}  
    public double getY() {return y;}  
    public String getName {return "Point";} }
```



```
public class Circle extends Point {  
    protected double radius;  
    public Circle() {super(0,0); setRadius(0) ;}  
    public Circle(double a, double b, double r)  
        {super(a,b); setRadius(r) ; }  
    public void setRadius (double r ){  
        radius = (r >= 0 ? r : 0); }  
    public double getRadius() {return radius;}  
    public double area() {return 3.14159*radius*radius;}  
    public String getName {return "Circle";} }  
}
```

```
public class Cylinder extends Circle{  
    protected double height; //height ofCylinder  
    .....  
}
```



```
interface CanFight {    void fight(); }
interface CanSwim{    void swim(); }
interface CanFly{    void fly(); }
class ActionCharacter { public void fight() { } }
class Hero extends ActionCharacter implements
    CanFight, CanSwim, CanFly {
    public void swim() { }
    public void fly() { }
}
public class Adventure {
    static void t(CanFight x) {x.fight();}
    static void u(CanSwim x) {x.swim();}
    static void v(CanFly x) {x.fly();}
    static void w(ActionCharacter x) {x.fight();}
    public static void main(String[] args) {
        Hero i = new Hero();
        t(i) ; u(i) ; v(i) ; w(i) ; }
}
```

- 一个类型是指一组值集及其上的操作集所构成的二元组

- 类的类型.值集: 该类及子类的所有实例对象.

操作集: 类中定义的所有方法

- 接口的类型.值集: 所有接口的实现类及子类的实例对象

操作集: 接口中定义的所有方法

- 若一个类实现了一个接口, 则该类及其子类所实例化的对象既属于这个类的类型, 也属于那个接口的类型, 即该对象可以作为这两种类型来引用。这是Java语言类型“多态”的一种形式



- 包的成员包括相互关联的类、接口和子包
 - Package穿建了一个相互关联的接口和类的群体
 - 需要命名管理机制
 - 控制成员访问
- **Java Package**
 - Package
 - Import



- `package pkg1[.pkg2[.pkg3]];`
- 如果没有package语句则系统会为每个. Java源文件创建一个无名包，但它不能被其他包所引用
- Java编译器根据文件系统来存储包中的类，所以包的结构也是树型结构
- Example: `package java.awt.image`

java/awt/image(unix)

java\awt\image(Win95), ect

- 编译类：使用全路径，或将package路径放入CLASSPATH(such as “.; \java\classes; <prodir>”)



```
package p1;  
Public class Protection { ...}  
package p2;  
public class Derived2 extends p1.Protection{...}
```

```
E:\javafile>md p1
```

```
E:\javafile\p1>edit (Protection.java)
```

```
E:\javafile>md p2
```

```
E:\javafile\p2>edit (Derived2.java)
```

```
E:\> Set classpath=e:\jdk\lib\classes.zip;e:\javafile;
```

```
E:\> e:\jdk\bin\javac e:\javafile\p1\Protection.java
```

```
E:\>e:\jdk\bin\java p1.Protection
```



- 原因
 - 所有内建类都被组织进 (java) packages
 - 在程序内使用完全路径名不是一个好的方法
 - ‘imported’ 可以直接访问包或类
- Syntax `import pkg1[.pkg2].(<classname>/*);`
- Example
 `import java.util.Date; import java.io.*;`
- Notes
 - 使用的类或包必须加载
 - 程序中使用 ‘*’ 将会延长编译时间，但不会影响类的大小和性能



- 控制数据或方法的可见性
- 范围
 - 类
 - 包
 - 访问控制符: `private`, `public`, `protected`, `package(default)`
- 两个类之间的关系
 - R1: 同一个包中的子类
 - R2: 同一个包中的非子类
 - R3: 不同包中子类
 - R4: 不同包中的非子类



```
package p1;  
Public class Protection {  
    int n =1;  
    private int npri = 2;  
    protected int npro = 3;  
    public int npub=4;  
    public Protection() {  
        System.out.println("base constructor");  
        System.out.println("n = " + n);  
        System.out.println("npri =" + npri);  
        System.out.println("npro =" + npro);  
        System.out.println("npub =" + npub); }  
    private void f1() {  
        System.out.println("f1-" + npri); }
```



```
Protected void f2() {  
    System.out.println("f2-" + npro); }  
Public void f3() {  
    System.out.println("f3-" + npub); }  
void f4() {  
    System.out.println("f4-" + n); }  
Public static void main(String args[]) {  
    Protection pp = new Protection();  
    System.out.println("pp");  
    pp.f1();    pp.f2();    pp.f3();    pp.f4() } }
```

base constructor**n=1****npri=2****npro=3****npub=4****pp****f1-2****f2-3****f3-4****f4-1**

```
package p1;
public class Derived extends Protection {
    public Derived() {
        System.out.println("derived constructor");
        System.out.println("n =" + n);

        System.out.println("npro =" + npro);
        System.out.println("npub =" + npub); }
    public void g() {

        f2();
        f3();
        f4();
    }
}
```




```
public static void main(String args[]) {  
    Protection pp = new Protection();  
    System.out.println("pp");  
  
    pp.f2();  
    pp.f3();  
    pp.f4();  
    Derived dd = new Derived();  
    System.out.println("dd");  
  
    dd.f2();  
    dd.f3();  
    dd.f4();  
    }  
}
```

base constructor、 n=1
npri=2、 npro=3 、
npub=4、 pp

f2-3、 f3-4、 f4-1

base constructor、 n=1
npri=2、 npro=3 、
npub=4、

derived constructor、
n=1、 npro=3 、
npub=4、 dd

f2-3、 f3-4、 f4-1



```
package p1;  
Public class Nderived {  
    public Nderived() {  
        System.out.println("nderived constructor");  
    }  
    private void ng() {  
    }  
}
```



```
public static void main(String args[]) {  
    Protection pp = new Protection();  
    System.out.println("pp");  
  
    pp.f2();  
    pp.f3();  
    pp.f4();  
    Derived dd = new Derived();  
    System.out.println("dd");  
  
    dd.f2();  
    dd.f3();  
    dd.f4();  
    }  
}
```

base constructor、 n=1
npri=2、 npro=3 、
npub=4、 pp

f2-3、 f3-4、 f4-1

base constructor、 n=1
npri=2、 npro=3 、
npub=4、

derived constructor、
n=1、 npro=3 、
npub=4、 dd

f2-3、 f3-4、 f4-1





— — — — —



Lay

```
public static void main(String args[]) {  
    p1.Protection pp = new p1.Protection();  
    System.out.println("pp");
```

```
    pp.f3();
```

```
    Derived2 ff = new Derived2();  
    System.out.println("ff");
```

```
    ff.f2();  
    ff.f3();
```

```
    }
```

```
}
```

base constructor、 n=1
npri=2、 npro=3 、
npub=4、 pp、 f3-4

base constructor、 n=1
npri=2、 npro=3 、
npub=4、

derived other package
constructor 、
npro=3 、 npub=4、
ff 、 f2-3、 f3-4



```
package p2;  
public class Nderived2 {  
    public Nderived2() {  
        System.out.println("derived other package constructor");  
    }  
  
    public void ng2() {  
    }  
}
```



```
public static void main(String args[]) {  
    p1.Protection pp = new p1.Protection();  
    System.out.println("pp");
```

```
    pp.f3();
```

```
    p1.Derived dd = new p1.Derived();  
    System.out.println("dd");
```

```
    dd.f3();
```

```
    }
```

```
}
```

base constructor、 n=1
npri=2、 npro=3 、
npub=4、 pp、 f3-4

base constructor、 n=1
npri=2、 npro=3 、
npub=4、

derived constructor、
n=1、 npro=3 、
npub=4、 dd 、 f3-4



Package pkg1;

B access
A's public
protected*
package

A access
A's public
protected
Private

Package pkg2;

D access
A's public

import pkg1.*;
Client D

Client B

C access
A's public
protected
package

Public ClassA
Public void f1()
Protected void f2()
Void f3()
Private void f4()

E access
A's public
protected

import pkg1.*;
Subclass E

Subclass C




```
Class B {  
    B() { };  
    public/protected/private/frendly void f() { }  
}  
Class D extends B {  
    D(){};  
    void g() { f(); };  
    public static void main(String args[]) {  
        B b = new B();  
        b.f();  
    }  
}
```



Modifier Relation of B&D	No Modifier	Public	Protected	Private
Derived & Same Package	Yes	Yes	Yes(same with C++)	No
Not Derived & Same Package	No	No	No	No
Derived & Different Package	No	Yes	Yes(same with C++)	No
Not Derived & Different Package	No	No	No	No



Modifier Relation of B&D	No Modifier	Public	Protected	Private
Derived & Same Package	Yes	Yes	Yes(different with C++)	No
Not Derived & Same Package	Yes	Yes	Yes	No
Derived & Different Package	No	Yes	No(same with C++)	No
Not Derived & Different Package	No	Yes	No	No



- P122 5-12

