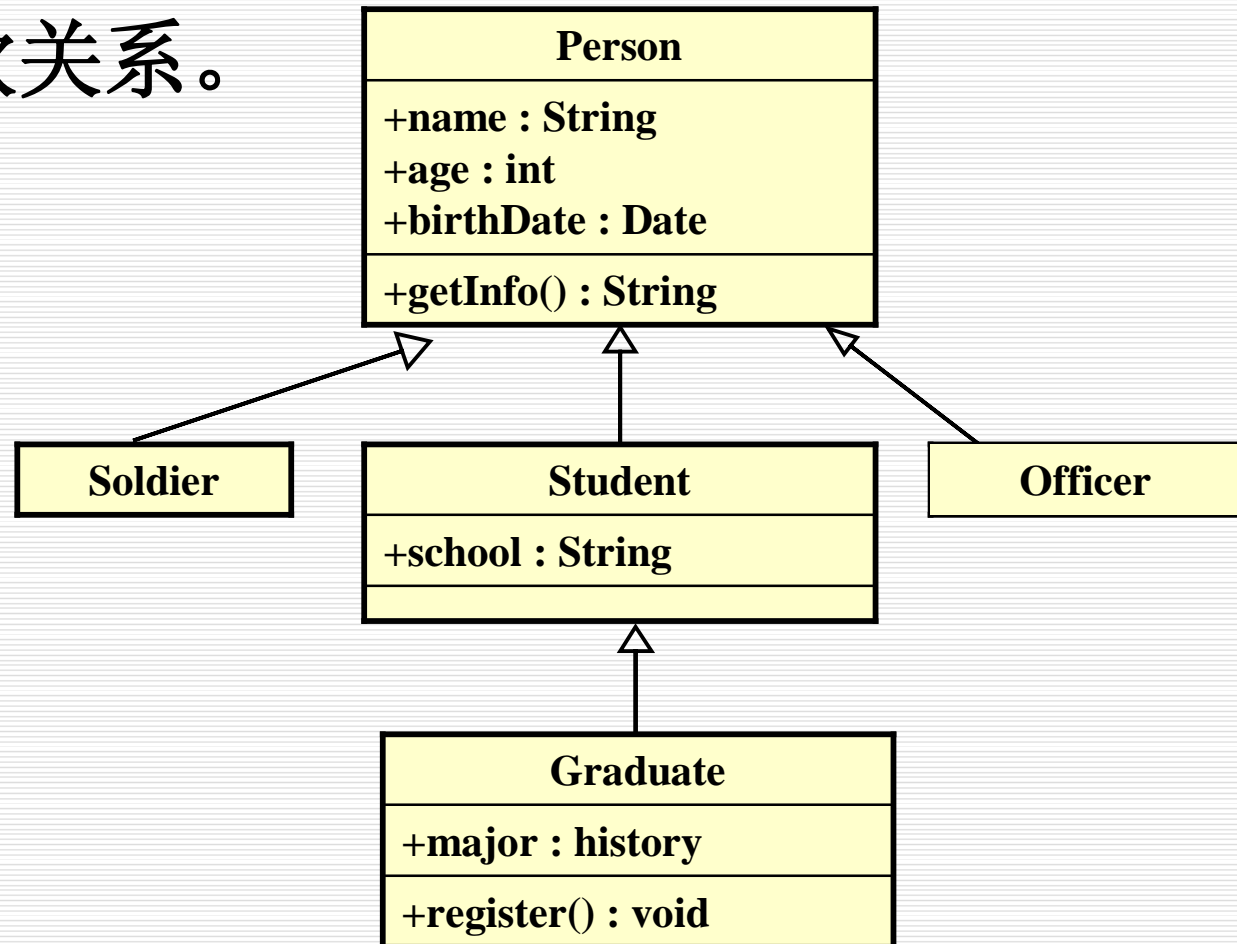


---

# 面向对象2

# 类的继承(inheritance)

客观世界中存在许多层次关系，类的继承反映了这种层次关系。



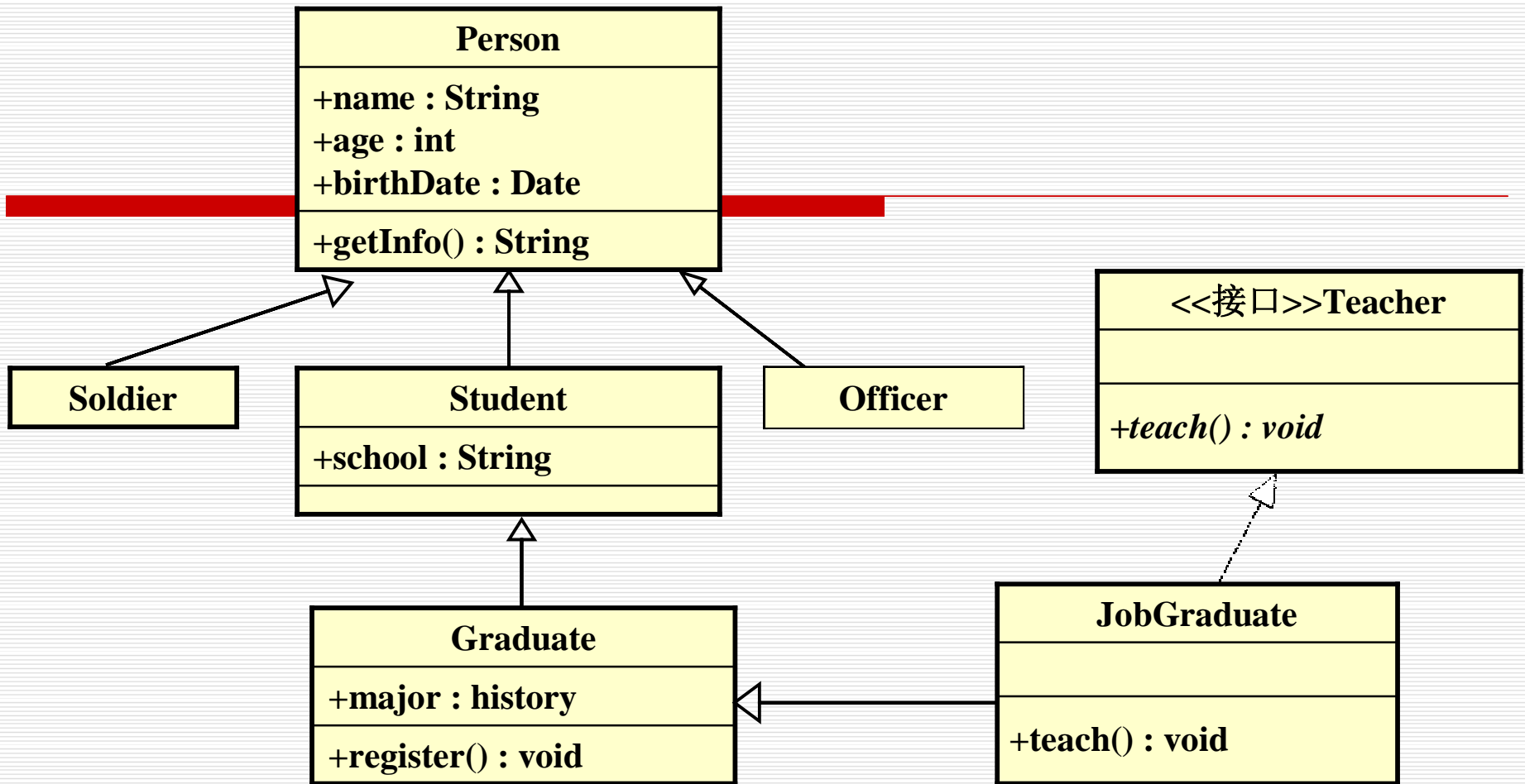
## 类的继承的特点

---

类继承也称为类派生，是指一个类可以继承其他类的**非私有成员**，实现代码复用。

被继承的类称为**父类**或**超类**，父类包括所有直接或间接被继承的类；继承父类或超类后产生的类称为**派生类**或**子类**。Java语言以**Object**类作为所有类的父类，所有的类都是直接或间接地继承Object类得到的。

**在Java语言中，只允许单继承**。所谓单继承是指每个类只有一个父类，不允许有多个父类。但一个类允许同时拥有多个子类，这时这个父类实际上是所有子类的公共成员变量和公共方法成员的集合，而每一个子类则是父类的特殊化，是对公共成员变量和方法成员的功能、内涵方面的扩展和延伸。



Java语言的多继承可通过接口来实现。

类继承不改变成员的访问权限，父类中的成员为公有的或被保护的，则其子类的成员访问权限也继承为公有的或被保护的。

## 类继承的实现

---

在Java语言中，扩展和继承机制是通过类定义中的 extends 子句实现的，其格式如下：

```
[<修饰符>] class <子类名> extends <直接超类名> {  
    [<成员变量定义>...]  
    [<构造方法定义>...]  
    [<初始化块>...]  
    [<方法定义>...]  
}
```

extends 子句指定了被扩展的类，称为当前定义类的直接超类，而当前定义类称为被扩展类的直接子类。通常把一个类A称为另一个类C的子类，是指满足下面条件之一者：

- 
- 类A是类C的直接子类。
  - 存在一个类B，类A是类B的子类，类B是类C的子类。

如果类A是类C的子类，那么类C称为是类A的超类。一个类不能把自己作为超类。

在类的定义中，如果缺省extends子句，则Object类被作为当前定义类的直接超类。在Java中，所有的类组成一个倒树型的层次结构，其中根是Object类。所以Object类是所有类的超类，而Object类本身没有超类。例如：

```
class X { int x; }  
class Y extends X { int y; }  
class Z extends Y { int z; }
```

## 继承例子

```
class A
{  int x = 2;
   private int y = x;
   int m1()
   { return x * y;  }
}
class B extends A
{  int z = ( x + 1 ) * 5;
   int m2()
   { return m1() * z;  }
}
class Test
{ public static void main(String args[ ])
  {  B o = new B();
     System.out.println("Result of m2()="+o.m2());
  }
}
```

结果:

Result of m2() =60

# 类成员

---

一个类除了自己在类体中定义的成员之外，还能够从其直接超类和直接超接口中继承一些成员。具体来说，类类型包括以下成员：

- 在类体中定义的成员。
- 从它的直接超接口继承的成员。
- 从它的直接超类继承的成员。

注：

**Object**类没有直接超类，所以不存在继承的问题。



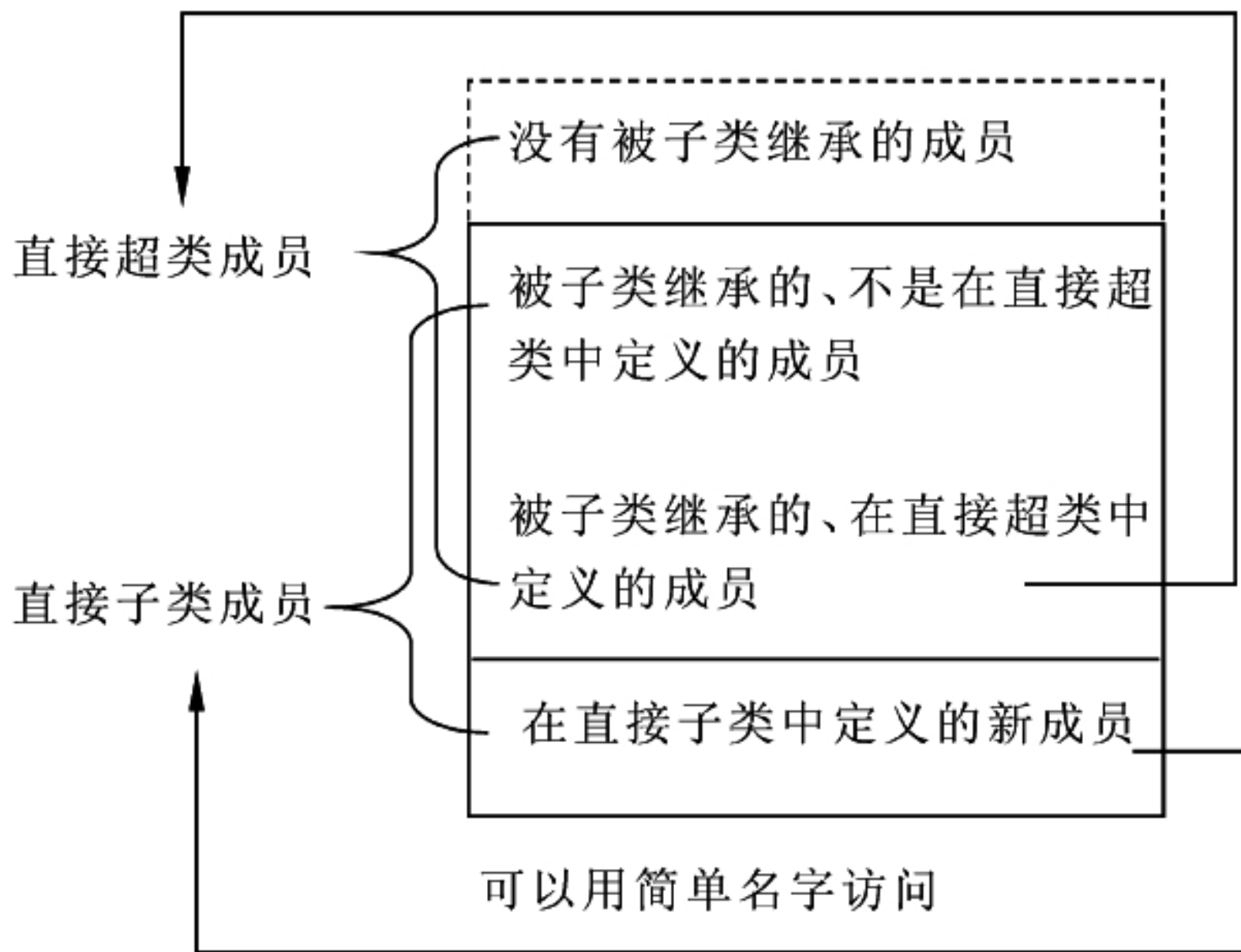
---

超类中声明为**private**的成员**不会被子类继承**。若超类和子类不在同一个包内，则超类中没有用**protected**、**public**修饰的成员不被会子类继承。

静态初始化块和构造方法都不是成员，因此也不会被继承。

从直接子类的角度来看，直接超类的成员可以被分为被子类继承和不被子类继承两大类。被子类继承的成员又可以被分为是在其类体中定义的和是从其超类中继承的两类。

可以用简单名字访问



## "is-a"关系

"is-a"关系是"特殊"—"一般"关系的具体体现，即一个"特殊"类型的个体同样是一个"一般"类型的个体。

拿前面的X、Y和Z类为例，可以用下面代码说明"is-a"关系：

`Z o1 = new Z();`//创建类Z的一个实例，并将对实例的引用值赋给类型为Z的变量o1

`Y o2 = o1;`//同样可以将实例的引用值赋给类型为Y的变量o2

`X o3 = o1;`//也可以将实例的引用值赋给类型为X的变量o3

即类Z的一个实例同样可以被看作是其超类Y或者X的一个对象。也可以反过来说明这种关系，即：一个定义为"类"类型的变量可以拥有某个对象的引用值，

这个对象是该类或者该类的任何子类的一个实例。

`X o = new X();` //类型为类X的变量o引用类X的一个实例

`o = new Y();` //变量o也可以引用类Y的一个实例

`o = new Z();` //变量o也可以引用类Z的一个实例

之所以可以将子类对象看作是超类对象，是因为**子类**一般比超类具有**更多的成员**。当把子类对象当作超类对象时，只需忽略那些多余的成员即可。实际情况也确实如此：**如果将一个子类对象引用值赋给一个超类类型变量，那么通过该超类类型变量就只能访问超类中旧的成员(实例方法有可能被覆盖)，而不能访问在子类中定义的新成员。**

反方向的赋值是不符合逻辑的，即一个超类实例的引用值是不能够赋给一个子类类型变量的。

## 父类对象与子类对象的转换

类似于基本数据类型之间的强制类型转换。存在继承关系的父类对象和子类对象之间也可以在一定条件之下相互转换。

这种转换需要遵守以下原则：

- ①子类对象可以被视为是其父类的一个对象
- ②父类对象不能被当作是某一个子类的对象。
- ③如果一个方法的形式参数定义的是父类对象，那么调用这个方法时，可以使用子类对象作为实际参数。
- ④如果父类对象的引用指向的实际是一个子类对象，那么这个父类对象的引用可以用强制类型转换转化成子类对象的引用。

## 成员变量隐藏

---

成员变量隐藏是指在子类中定义了一个与直接超类中的某个**成员变量同名**的成员变量，从而使直接超类中的那个成员变量不能被子类继承。这里不管隐藏的和被隐藏的变量是否都为实例变量或者类变量，也不管他们的类型是否相同。

当出现成员变量隐藏时，在超类类体代码中用简单变量名访问的一定是超类中的成员变量，而在子类类体代码中用简单变量名访问的则一定是定义在子类中的成员变量。另外，可以用下列格式访问超类中被隐藏的成员变量：

## 调用隐藏的超类成员

---

○ **super.**<变量名> //在**子类类体内**，访问直接超类中被隐藏的成员变量

○ **((<超类名>)<子类实例引用>).**<变量名> //访问指定超类中被隐藏的成员变量

○ **<超类名>.**<变量名> //访问指定超类中被隐藏  
的类变量

关键字**super**和**this**都表示当前对象的引用，不同的是：**this**的编译期类型为方法体所在的类，而**super**的编译期类型则为当前类的直接超类。**super**和**this**都应该在类的实例方法中使用。

## 类型转换举例

```
class X
{   int x=1;
    void method1()
    {   System.out.println("class X:"+"x="+x); }
}

class Y extends X
{   int y=2;
    void method2()
    {   System.out.println("class Y:"+"x="+x
                            +" y="+y);
    }
}
```



```
class XYZ_Demo
```

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

```
{ X o1=new X(); //o1为父类实例
```

```
o1.method1();
```

```
Y o2=new Y(); //o2为子类实例
```

```
o2.method1();
```

```
o2.method2();
```

```
X o=o1; //o为父类实例
```

```
o.method1();
```

```
o=o2; //类型转换-上溯造型
```

```
o.method1();//o.method2()父类无method2方法
```

```
((Y)o).method2();//子类实例赋给父类实例,想
```

```
} //调用子类的方法，就要转化
```

```
}
```

```
class X:x=1
```

```
class X:x=1
```

```
class Y:x=1 y=2
```

```
class X:x=1
```

```
class X:x=1
```

```
class Y:x=1 y=2
```

```
class X{ int x=1; }
```

成员变量隐藏举例

```
class Y extends X{ int x=11; int y=2; }
```

```
class Test{
```

```
void method2()
```

```
{ System.out.println("class Y:"+"x="+x+" y="+y);
```

```
System.out.println("class Y:"+"x="+x+" y="+y+"  
x="+super.x); }
```

```
System.out.println("x="+o2.x+" y="+o2.y  
+" x="+((X)o2).x);
```

```
o1=o2;
```

```
System.out.println("x="+o1.x);
```

```
}
```

```
}
```

```
Y o2=new Y();
```

```
o2.method2();
```

x=1

x=11 y=2

x=11 y=2 x=1

x=1

```
class Example0502
```


成员变量隐藏举例

```
{ int x;  
void set(int a)  
{ x = a; }  
void print()  
{ System.out.println("x = " + x); }  
}
```

```
class Example0502_1 extends Example0502
```

```
{ int x;  
void newset(int a)  
{ x = a; }  
void newprint()  
{ System.out.println("x = "+x+" "+super.x); }  
}
```

```
class Test
{ public static void main(String args[])
  { Example0502_1 o = new Example0502_1();
    o.set(10);
    o.newset(100);
    o.print();
    o.newprint();
    System.out.println("x="+o.x+" "+
                        ((Example0502)o).x);
  }
}
```

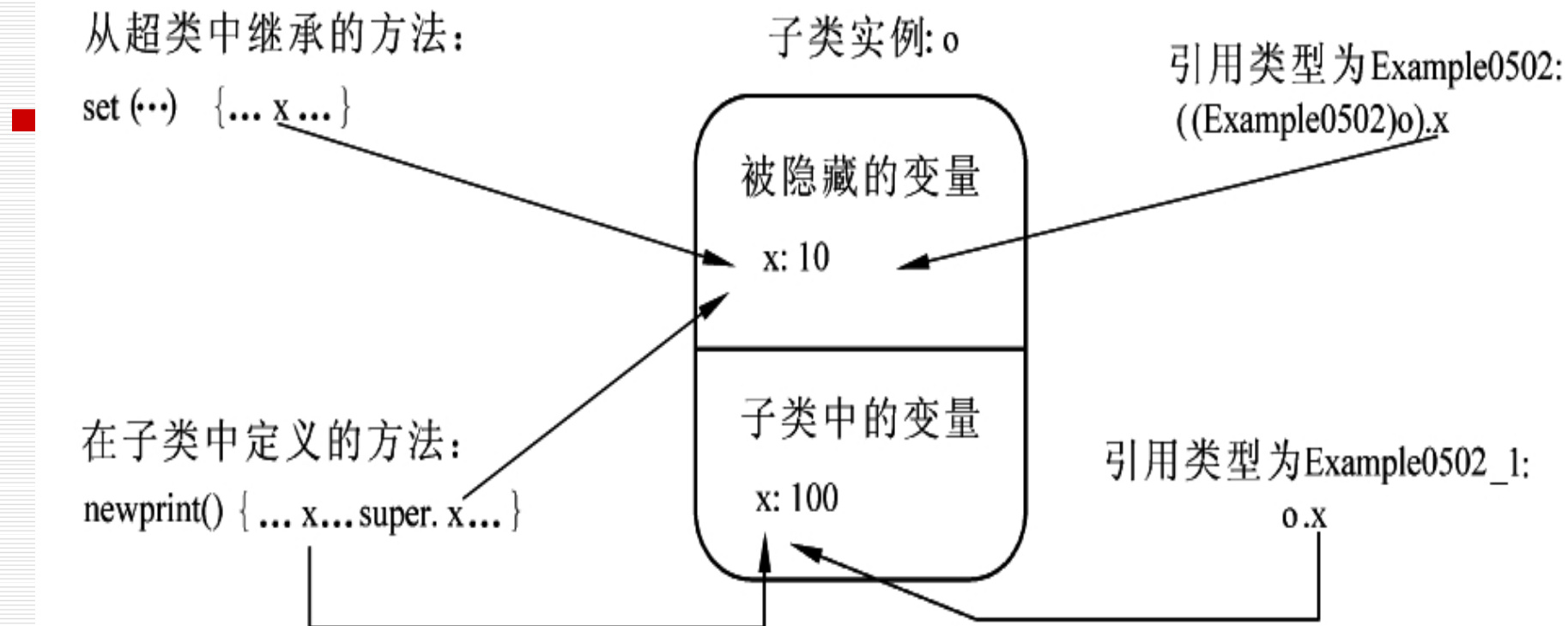


程序的输出结果为:

**x = 10**

**x = 100 10**

**x = 100 10**



例题示意图(变量隐藏)

程序的输出结果为:

**x = 10**

**x = 100 10**

**x = 100 10**

## 成员方法覆盖

---

子类也可以定义与父类同名的方法，实现对父类方法的覆盖。即应与父类有完全相同的**方法名、返回值和参数列表**。应符合下列规则：

- 1 覆盖方法不能比它所覆盖的方法访问性差。
- 2 覆盖方法不能比它所覆盖的方法抛出更多的异常。

方法成员的覆盖与成员变量的隐藏的不同：子类隐藏父类的成员变量只是使得它不可见，父类的同名成员变量在子类对象中仍然占据自己的存储空间；而子类成员方法对父类同名方法的覆盖将清除父类方法占用的内存空间，从而使得父类的方法在子类对象中不复存在。

# 注意

---

- 1 覆盖(overriding)与重载(overload)的区别
- 2 static方法不能被覆盖(子类可定义)
- 3 被final修饰的方法不能被覆盖,保证程序的安全性和正确性 (子类不能定义)
- 4 static、final修饰的成员变量可以被隐藏

---

当出现方法覆盖时，如果要在子类中访问直接超类中被覆盖的方法，使用包含关键字super的方法访问表达式，即： **super**.<方法名>([<实参表>])

无法通过提升实例引用的类型来访问超类中被覆盖的方法。这是方法覆盖和成员变量隐藏的区别所在。当采用下面格式调用方法时：

((<超类名>)<子类实例引用>).<方法名>([<实参表>])

仍调用子类的方法!!!



---

系统首先会在编译时检查被调用的方法是否为指定超类的成员。而在运行时，系统则调用子类中相应的成员方法。

当超类类体代码中出现用简单名字调用其成员方法时，程序运行时调用的不见得就一定是超类中的成员方法。如果子类中定义有覆盖方法，而调用的主体又是子类实例，那么实际调用的就会是子类中定义的覆盖方法。

## 方法覆盖举例

```
class Parent
{   void test()
    { System.out.println("parent"); }
    void print1()
    { test(); } }

class Child extends Parent
{   void test()
    { System.out.println("child"); }
    void print2()
    { test(); } }

class Test
{   public static void main(String args[])
    {   Child o = new Child();
        o.print2();
        o.print1();    //调用哪个test()方法?
    }
}
```

child  
child

## 方法覆盖举例

```
class X
{   int x=1;
    void method1()
    {   System.out.println("class X:"+"x="+x); }
}

class Y extends X
{   int y=2;
    void method1()
    {   System.out.println("class Y:"+"x="+x+
                            "y="+y);
        //super.method1();
    }
}
```

```
class Test{  
    public static void main(String[] args){  
        X o1=new X();  
        o1.method1();  
        Y o2=new Y();  
        System.out.println("x='"+o2.x+" y='"+o2.y);  
        o2.method1();  
        ((X)o2).method1();  
        o1=o2;  
        o1.method1();  
    }  
}
```

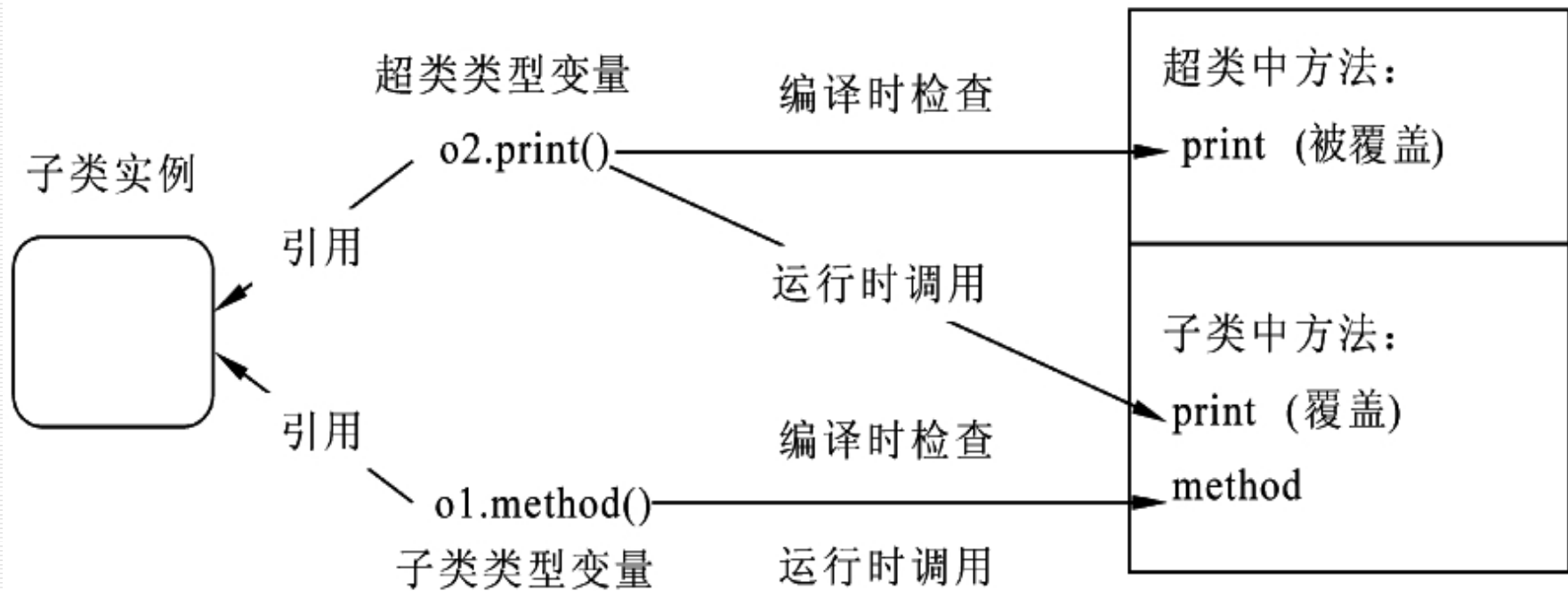
```
class X:x=1  
x=1 y=2  
class Y:x=1 y=2  
class Y:x=1 y=2  
class Y:x=1 y=2
```

## 方法覆盖举例

```
1) class Example0503
2) { void print()
3)   { System.out.println("superclass");
4)   }
5) }
7) class Example0503_1 extends Example0503
8){ void print()
9)   { System.out.println("subclass"); }
11) void method()
12) { print();           //调用所在类成员方法
13)   super.print(); //调用直接超类中的被覆盖方法
14) }
15) }
```

```
17)class Test {  
18) public static void main(String args[])  
19) { Example0503_1 o1=new Example0503_1();  
20)     o1.method();  
21)     Example0503 o2 = o1;  
22)     o2.print();  
23) }  
24)}
```

程序的输出结果是：  
subcl ass  
supercl ass  
subcl ass



例题示意图(方法覆盖)

## 方法覆盖与变量隐藏的比较

```
1) class Point {  
2)     int x = 1, y = 1;  
3)     void move(int dx, int dy) {  
4)         x += dx;  
5)         y += dy;  
6)     }  
7)     void method() {  
8)         x += 5; y += 5;  
9)         move(10, 10); //调用哪个move?  
10)    }  
11) }
```



12)class RealPoint **extends** Point

13){ double x=2.0, y=2.0;

14) void **move**(int dx, int dy)

15) { x += dx; y += dy; }

18)}

19)class Test

20){ static void show\_1(**RealPoint** p)

21) { System.out.print("方法show\_1输出:");

22) System.out.println(p.x+", "+p.y); }

24) static void show\_2(**Point** p)

25) { System.out.print("方法show\_2输出:");

26) System.out.println(p.x+", "+p.y); }

```
28) public static void main(String[] args)
29) { RealPoint p = new RealPoint();
30)   show_1(p);           方法show_1输出: 2.0, 2.0
31)   show_2(p);           方法show_2输出: 1, 1
32)   p.move(10, 10);
32)   ((Point)p).move(10,10);//调用谁的move方法?
33)   show_1(p);           方法show_1输出: 22.0, 22.0
34)   show_2(p);           方法show_2输出: 1, 1
35)   p.method();
36)   show_1(p);           方法show_1输出: 32.0, 32.0
37)   show_2(p);           方法show_2输出: 6, 6
38) }
39)}
```

## 派生类的初始化

---

每当创建类的一个实例时，类中至少会有一个构造方法被隐含调用。如果一个类没有定义构造方法，那么系统会自动提供一个默认的构造方法；反之，如果一个类定义了构造方法，系统就不再提供默认的构造方法。

若类中存在多个构造方法，一个构造方法体中可以显式调用另一个构造方法，格式为：

```
this( [<实参表> ] );
```

注意：不能通过该形式直接或间接地调用自己，否则将会引发编译时错误。

---

构造方法体中的**第一条语句**也可以是对直接超类中的一个构造方法的显式调用，格式为：

**super( [<实参表> ] );**

如果当前类不是Object类，而某个构造方法体的**第一条语句不是上述对本类或其直接超类的另一个构造方法的显式调用**，那么系统会隐含地假定该构造方法体由语句**super();**开始。

## 构造方法调用举例

```
class Example0505
```

```
{ int i;
```

```
    Example0505()
```

```
    { i = 10; }
```

```
}
```

```
class Example0505_1 extends Example0505
```

```
{ int j, k;
```

```
    //隐含调用直接超类中不带形参的构造方法
```

```
    Example0505_1()
```

```
    { j = 20; }
```

```
    Example0505_1(int a)
```

```
    { this(); //调用当前类中不带形参的构造方法
```

```
        k = a; }
```

```
}
```

---

```
class Test
```

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

```
    { Example0505_1 o = new Example0505_1(30);
```

```
        System.out.println("i="+o.i + "j=" + o.j  
                             + "k="+o.k);
```

```
    }
```

```
}
```

下面是该程序的输出结果:

i = 10   j = 20   k = 30

---

在创建子类的对象时，使用子类的构造方法对其初始化，不但要对自身的成员变量赋初值，还要对父类的成员变量赋初值。因为成员变量赋初值通常在构造方法中完成，而且父类的构造方法不能被继承，因此在Java语言中，允许派生类调用父类的构造方法。

---

派生类的初始化遵循如下的原则：

(1) 若子类无自己的构造方法，则它将调用父类的无参构造方法；若子类有自己的构造方法，则在创建子类对象时，它将先调用父类的无参构造方法，然后再执行自己的构造方法。

(3) 若父类只有有参数的构造方法，子类必须通过在自己的构造方法中使用super关键字来调用它，但这个调用语句必须是子类构造方法的第一个可执行语句。



```
class Animal
{
    Animal()
    {   System.out.println("Animal's constructor ");   }
    Animal(int num)
    {   System.out.println("Animal's constructor="+num);   }
}
class Mankind extends Animal
{
    Mankind()
    {   System.out.println("Mankind's constructor ");   }
    Mankind(int num)
    {   System.out.println("Mankind's constructor="+num);   }
}
```

```

public class Kids extends Mankind
{
    Kids()
    {   System.out.println("Kids' constructor");   }
    Kids(int num)
    {   super(999); //super()?
        System.out.println("Kids' constructor="+num); }
    public static void main(String[] args)
    {
        Kids kid1=new Kids();
        System.out.println("Animal's constructor");
        Kids kid2=new Kids(999);
        System.out.println("Mankind's constructor=999");
    }
}

```

```

Animal's constructor
Mankind's constructor
Kids' constructor
-----
Animal's constructor
Mankind's constructor=999
Kids'constructor=999

```

1) **class Example0506**

2) { **int i = 10;**

3) **int j=20;**

4) **Example0506(int a)**

6) { **j = a;**}

7) }

9) **class Example0506\_1 extends Example0506**

10){ **int k = j + 10, l;**

11) **Example0506\_1()**

12) { **super(5);**

13) **l = k + 5;**

14) }

15) }

①

②

③

```
17) class Test {  
18)   public static void main(String args[]){  
19)     Example0506_1 o = new Example0506_1();  
20)     System.out.print(" i="+o.i+" j="+o.j);  
21)     System.out.print(" k="+o.k+" l="+o.l);  
22)   }  
23) }
```

该程序的输出结果是:

**i = 10 j = 5 k = 15 l = 20**

## 关键字 `null`、`this` 和 `super`

---

Java系统默认：每个类都缺省地具有 `null`、`this` 和 `super` 三个域，可以在任意类中不加说明地直接使用这三个域。

其中：

- (1) `null` 代表"空"值，在定义一个对象但尚未为其开辟内存空间时，可以指定这个对象为 `null`。
- (2) `this` 表示的是当前对象本身，它实际代表了当前对象的一个引用。通过引用可以顺利地访问到对象，包括访问、修改对象的域、调用对象的方法。一个对象可以有若干个引用，`this` 只是其中的一个。

---

(3) **super**表示的是当前对象的**直接父类对象**，  
是对直接父类对象的引用。

(4)应注意的问题：

**this** 和 **super** 是属于类的有特指的域，只能用来代表当前对象或当前对象的父对象，而不能像其它类的属性随意引用。

# this与super的用法

---

this代表当前对象的一个引用。

有三种使用场合：

1. 用来访问当前对象的数据成员，使用形式：

**this. 数据成员**

2. 用来调用当前对象的成员方法，使用形式：

**this. 成员方法**

3. 当有重载的构造方法时，用来引用同类的其他构造方法，使用形式：**this(参数)**

# this与super的用法

super代表当前对象的直接父类。

也有三种使用场合：

- 1 用来访问当前对象直接父类隐藏的数据成员，使用形式：

super. 数据成员

- 2 用来调用当前对象直接父类中被隐藏的成员方法，使用形式：

super. 成员方法

- 3 用来调用直接父类的构造方法，使用形式：

super(参数)



```
class ThisDemo
{ int x1=1, x2=2;
  void ss( )
  { int x1=10, x2=20;
    x1=this.x1;
    System.out.println("x1="+x1+"x2="+x2);
  }
}

public class abc2
{ public static void main(String args[])
  { ThisDemo td=new ThisDemo( );
    td.ss();
  }
}
```

```
class AddClass{  
    public int x,y,z;  
    AddClass() {}  
    AddClass(int x)  
    { this.x=x; }  
    AddClass(int x, int y)  
    { this(x); this.y=y; }  
    AddClass(int x, int y, int z)  
    { this(x,y); this.z=z; }  
    public int add( )  
    { return x+y+z; }  
}
```

```
public class Test extends AddClass  
{  
    public static void main(String arg[]){  
        AddClass p1=new AddClass(2,3,5);  
        AddClass p2=new AddClass(10,20);  
        AddClass p3=new AddClass(1);  
        System.out.println("x+y+z="+p1.add());  
        System.out.println("x+y="+p2.add());  
        System.out.println("x="+p3.add());  
    }  
}
```

```
class SuperClass  
{ int x;  
  SuperClass()  
  {  
    x=3;  
    System.out.println("in superClass:x="+x);  
  }  
  void doSomething( )  
  {  
    System.out.println("in superClass:doSomething()");  
  }  
}
```

```

class SubClass extends SuperClass
{
    int x;
    SubClass()
    { super(); //不加这句是什么效果?
      x=5;
      System.out.println("in subClass:x="+x);
    }
    void doSomething()
    { super.doSomething();
      System.out.println("in subClass:
                          doSomething()" );
      System.out.println("super.x="+ super.x+
                          "sub.x="+ x);    }
    }
}

public class Test
{
    public static void main(String args[ ])
    {
        SubClass subC=new SubClass( );
        subC.doSomething( );
    }
}

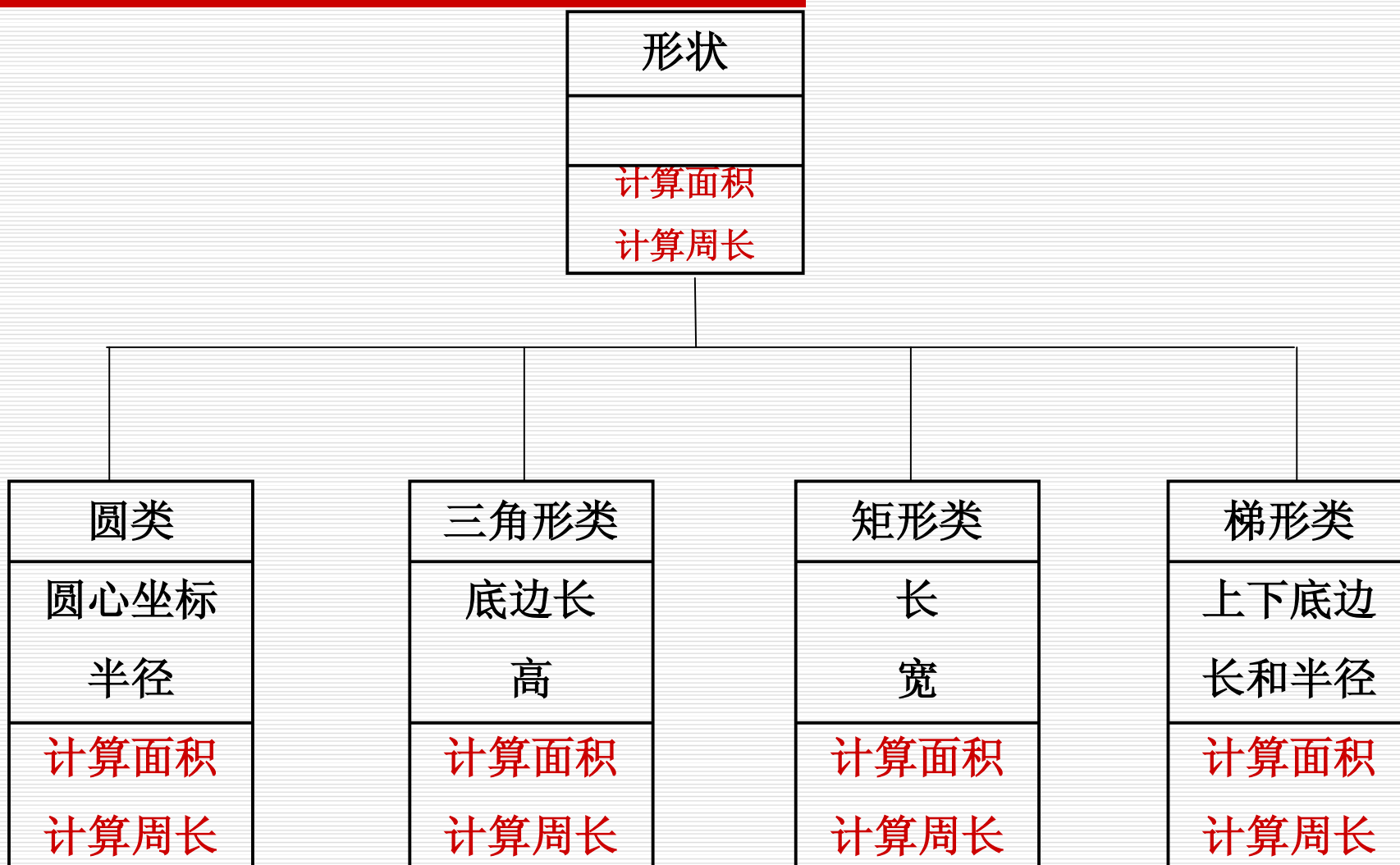
```

```

in superClass:x=3
in subClass:x=5
in superClass:doSomething()
in subClass:doSomething()
super.x=3 sub.x=5

```

# 抽象类和抽象方法



## 抽象类和抽象方法

---

- 1 抽象方法是只有返回值类型、方法名、方法参数而**不定义方法体**的一种方法。
- 2 抽象类是abstract修饰的类，至少含有一个抽象方法。
- 3 抽象类需要子类继承，在子类中实现抽象类中的抽象方法。
- 4 抽象方法必须定义在抽象类中。抽象类是一种未实现的类，抽象类不能用new实例化一个对象。
- 5 抽象类含有构造方法。

```
abstract class Shape
{ // 定义抽象类Shape和抽象方法display
    abstract void display();
}

class Circle extends Shape
{ void display() // 实现抽象类的方法
    { System.out.println("Circle"); }
}

class Rectangle extends Shape
{ void display() // 实现抽象类的方法
    { System.out.println("Rectangle"); }
}
```



```
class Triangle extends Shape  
{ void display() // 实现抽象类的方法  
    { System.out.println("Triangle"); }  
}
```

```
class Test  
{ public static void main(String args[]  
    { Shape obj;  
        obj=new Circle();    obj.display();  
        obj=new Rectangle(); obj.display();  
        obj=new Triangle();  obj.display();  
    }  
}
```

---

抽象类是通过方法的覆盖来实现程序多态性的，主要体现数据抽象的思想。

各子类继承父类的抽象方法后，形成了若干名字相同，返回值相同，参数列表也相同，目标一致但具体实现方法有差别。

接口是实现多重继承的唯一途径。

# 接口

---

接口是若干完成某一特定功能的没有方法体的方法（抽象方法）和常量的集合。

接口是一种特殊的类。

抽象类	接口
定义成员变量	定义常量
定义成员（抽象）方法	定义抽象方法
有构造方法	没有构造方法

# 接口的定义

---

接口的定义格式如下：

**[modifier] interface 接口名 [extends 父接口名]**

**{ // interfaceBody }**

接口定义包括两个方面的内容：定义**接口名**和**接口体**。接口体同抽象类相似，为常量和抽象方法的集合。

变量：**public final static**和**final static**(默认)

方法：**public abstract** (默认)

和抽象类相似，接口含有抽象方法，因此接口不能直接创建对象。

# 接口的实现

---

和抽象类相似，用派生类在实现接口。  
比较：

1 抽象类用 **extends** 来派生子类

接口用 **implements** 来实现(派生子类)

2 抽象类与一般类一样，只能使用 **单继承**

接口实现了 **多重继承**

相同：通过对抽象方法的 **覆盖** 来定义方法体。

【例5.17】接口的实现。

```
interface IfRect // 定义接口
```

```
{ double w=3, l=4;
```

```
void compute(); //public abstract void
```

```
}
```

```
class Crect implements IfRect //定义实现接口的类
```

```
{ public void compute()
```

```
{ System.out.println("长方形面积为: "+w*l); }
```

```
}
```

```
public class InterfaceDemo //定义主类,创建接口类对象
```

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

```
{ Crect ct= new Crect();
```

```
ct.compute();
```

```
}
```

```
}
```

程序运行结果如下：  
长方形面积为： 12.0

## 接口的继承和组合

---

接口可以通过关键字 **extends** 继承 **其他接口**。子接口将继承父接口中所有的常量和抽象方法。此时，子接口的非抽象派生类不仅需实现子接口的抽象方法，而且需实现继承来的抽象方法。

例如：

```
interface IfB extends IfA
```

```
interface IfA //定义接口IfA
{ String a = "在接口A中"; //变量默认为public final static
  void showA(); //方法默认为public abstract
}
interface IfB extends IfA //定义接口IfB, 它继承接口IfA
{ String b = "在接口B中";
  void showB();
}
interface IfC extends IfB //定义接口IfC, 它继承接口IfB
{ String c = "在接口C中";
  void showC();
}
```



// 定义实现接口IfC的类

class **MyClass** **implements** **IfC**

{ // 实现public方法，不带public则报错

**public** void showA(){ System.out.println(a); }

**public** void showB(){ System.out.println(b); }

**public** void showC(){ System.out.println(c); }

}

public class Test

{ public static void main(String args[])

{ MyClass mc=new MyClass();

mc.showA();mc.showB();mc.showC();

//System.out.println(IfA.a);

}

}

程序运行结果如下：  
在接口**A**中  
在接口**B**中  
在接口**C**中

## 接口的多态

接口使方法的描述说明和方法功能的实现分开考虑。

【例5.19】定义接口并实现接口，说明接口的多态。

```
interface OneToN
```

```
{ int disp(int n); }
```

```
class Sum implements OneToN //继承接口
```

```
{ public int disp(int n) //实现接口中的disp方法
```

```
    { int s=0, i;
```

```
        for(i = 1;i <= n;i ++){s += i;}
```

```
        return s;
```

```
    }
```

```
}
```

```
class Pro implements OneToN // 继承接口
{ public int disp(int n) // 实现接口中的disp方法
  { int m=1, i;
    for(i = 1;i <= n;i ++){ m *= i; }
    return m;
  }
}
```

程序的运行结果如下：  
1至n的和 = 55  
1至n的积 = 3628800

```
public class UseInterface
{ public static void main(String args[])
  { int n = 10;
    Sum s = new Sum();
    Pro p = new Pro();
    System.out.println("1至n的和 = " + s.disp(n));
    System.out.println("1至n的积 = " + p.disp(n));
  }
}
```

# 接口类型的使用

---

接口可以作为一种**类型**来使用。

在Java语言中，任何实现接口的类的实例都可以存储在该接口类型的变量中。通过这些变量可以访问类所实现的接口中的方法。Java运行时系统动态地确定应该使用哪个类中的方法。

## 【例5.20】接口类型的使用

---

```
public class UseInterface1
{ public static void main(String args[])
{   int n = 10;
    OneToN otn;
    Sum s = new Sum();
    otn = s; // 在接口类型变量中存储Sum类的实例
    System.out.println("1至n的和 = " + otn.disp(n));
    Pro p = new Pro();
    otn = p; // 在接口类型变量中存储Pro类的实例
    System.out.println("1至n的积 = " + otn.disp(n));
}
}
```

# 包 (package)

一组相关的类和接口集合称为包(类名空间)

包将java语言的类和接口有机地组织成层次结构, 这个层次结构与具体的文件系统的目录树结构层次一致。

## 5.4.1 创建包

包由包语句package创建, 其语法格式如下:

```
package [package1[.package2[.[...]]]]
```

指明源文件中定义的和接口属于哪个包

关键字package后的package1是包名,在package1下允许有次一级的子包package2, package2下可以有更次一级的子包package3等等。各级包名之间用"."号分隔。通常情况下, 包名称的元素被整个地小写。

---

**【例5.21】** 在一个名为Rect表示长方形的类的类定义前加语句package创建包。

```
package ch05; // Rect.java文件名  
class Rect {  
    double length;  
    double width;  
}
```

编译程序完成生成Rect.class文件后，可将当前目录的Rect.class文件复制或移动到创建的ch05子目录中。

## 5.4.2 使用包

在Java程序中，若要用到某些包中的类或接口，一种方法是在程序的开始部分写出相应的引入（**import**）语句，指出要引入哪些包的哪些类。另一种方法不用引入语句，直接在要引入的类和接口前给出其所在包名。

### 1.使用import语句

**import**语句的格式与意义如下：

// 引入PackageName包中的类和接口

**import PackageName.Identifier;**

// 引入PackageName包中的全部类和接口

**import PackageName.\*;**



---

## 2.直接使用包

一般用在程序中引用类和接口次数较少的时候，在要引入的类和接口前直接给出其所在包名。例如：

```
java.applet.Applet ap = new java.applet.Applet();
```

## 3.使用CLASSPATH环境变量

**CLASSPATH**环境变量的作用为当一个程序找不到它所需要的其他类的.class文件时，系统会自动到**CLASSPATH**环境变量所指明的路径中去查找。

利用例5.1定义的类Rect，计算长和宽分别为10和20的长方形面积。

```
package ch05;
```

```
//import ch05.Rect; 同一目录可不用导入，直接使用
```

```
class RectDemo
```

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

```
{ Rect rect1 = new Rect();
```

```
    double area;
```

```
    rect1.width = 10;
```

```
    rect1.length= 20;
```

```
    area = rect1.width * rect1.length;
```

```
    System.out.println("长方形面积是:" + area);
```

```
}
```

```
}
```

编译完成产生class文件后将其从当前目录复制或移动到ch05子目录下，可在**当前目录下**用如下的命令来执行：

```
java ch05.RectDemo
```

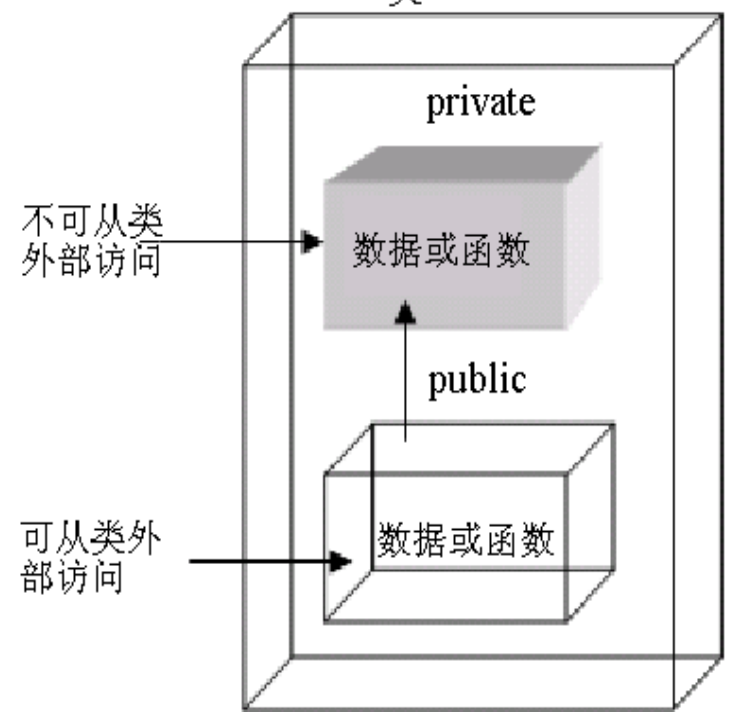
# 类成员的访问权限

包也是一种封装机制。包、类和访问修饰符共同形成了Java的访问控制机制。

没有用public修饰的类局限在包内使用。用private修饰的成员局限在类内使用。沿右田访问修饰符修饰的成员局限在包内(当前目录)

Java将类的成员可见性（可况，按照可见性的范围大小从

- (1)仅在本类内可见
- (2)在本类及其子类可见
- (3)在同一包内可见
- (4)在同一包内及其子类（不同
- (5)在所有包内可见



# 类的访问权限

类通常只用两种访问权限：缺省和**public**。

类声明为**public**时，可以被任何包的代码访问；缺省时，可被本包的代码访问。

类成员的可访问性如下：

访问级别 访问位置	<b>private</b>	无修饰符	<b>protected</b>	<b>public</b>
同类	√	√	√	√
同包，子类		√	√	√
同包，非子类		√	√	√
不同包，子类			√	√
不同包，非子类				√

# Protected例

---

// In the file: ClassA.java

package test1;

public class ClassA

{ **protected** int **proI** = 5;

**protected** void show()

{

    System.out.println("proI = " + proI);

}

}

```
// In the file: Demo.java
package test2;
import test1.ClassA;
class ClassB extends ClassA
{
    void showData()
    {
        show(); //不同包，子类可访问
    }
    void changeData(int newValue)
    {
        proI = newValue;
    }
}
class Test
{
    public static void main (String args[])
    {
        ClassB b = new ClassB();
        b.showData();
        b.changeData(10);
        b.showData();
    }
}
```

**proI = 5**  
**proI = 10**

# private例

---

**// In the file: ClassA.java**

**package test;**

**public class ClassA**

**{ private int proI = 5;**

**private void show()**

**{**

**System.out.println("proI =" + proI);**

**}**

**}**

//In the file: Demo.java

**import test.ClassA;**

**class ClassB extends**

**{ void showData()**

**{ show(); }**

**void changeData()**

**{ proI = new Value**

**}**

**class Test**

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

**{ ClassB b = new ClassB();**

**b.showData();**

**b.changeData(10);**

**b.showData();**

**}**

**}**

Demo.java:4: 找不到符号

符号: 方法 show()

位置: 类 ClassB

^

Demo.java:6: proI 可以在 test.ClassA 中访问

private

**{ proI = new Value; }**

^

2 错误



# 方法成员的覆盖与成员变量的隐藏

---

不同之处在于：

- 子类隐藏父类的成员变量只是使得它不可见，父类的同名成员变量在子类对象中仍然占据自己的存储空间；
- 子类成员方法对父类同名方法的覆盖将清除父类方法占用的内存空间，从而使得父类的方法在子类对象中不复存在。

---

Java语言的**多态**是指程序中**同名的不同方法共存**的情况。

Java语言可通过**两种方式实现多态**。

- 1 通过**子类对父类方法的覆盖实现多态**；
- 2 利用**重载在同一个类中定义多个同名的不同方法来实现多态**。

回顾:

```
class Parent
```

```
{ int x=1,y=2;
```

```
    int method()
```

```
    { return (x+y); }
```

```
}
```

```
class Child extends Parent
```

```
{ int x=3,y=4,z=5;
```

```
    int method()
```

```
    { return (x+y); }
```

```
}
```

```
public class Test2
{
    public static void main (String[] args)
    {
        Parent p=new Parent();
        Child c=new Child();
        System.out.println(p.x + " " + p.y);
        System.out.println(p.method());
        System.out.println(c.x + " " + c.y);
        System.out.println(c.method());
        p=c; //上溯造型      c=p; ✗
        System.out.println(p.x + " " + p.y);
        System.out.println(p.method());
        c=(Child)p;
    }
}
```

---

下课! 😊

*Thank you!*