第三章 类的重用

3.1 类的继承

3.1.1 继承的概念

根据已有类定义新类，新类拥有已有类的所有功能。Java 只支持单继承，每个子类只能有一个直接超类。

超类是所有子类的公共属性及方法集合，子类则是超类的特殊化。继承机制可以提高程序的抽象程度，提高代码的可重用性。

子类对象

·从外部来看，包括：

（1）与超类相同的接口；

（2）可以具有更多的方法和数据成员；

·其内包含着超类的所有变量和方法

3.1.2 类基础的语法

[ClassModifier] class ClassName extends SuperClassName

{

//类体

}

类继承举例：

设有三个类：Person，Employee，Manager。让 EMPLOYEE 继承 person（因为前者相比后者来说是一个更具体的类），manager 继承 employee（因为前者相比后者来说是一个更具体的类）。

关于访问从超类继承的成员的权限：

子类不能直接访问从超类中继承的私有属性及方法，但可以使用公有（及保护）方法来进行访问。

例如：

```
public class B{
    public int a = 10;
    private int b =20;//私有成员
    protected int c = 30;//保护成员
    public int getB(){return b;}//实际上这是通过公有方法访问 b
}

public class A extends B{
    public int d;
```

```
    public void tryVariables(){
        System.out.println(a);//可访问
        System.out.println(b);//不可访问
        system.out.println(c);//可访问
        System.out.println(getB());//可以访问
}
```

3.1.3 隐藏和覆盖

隐藏和覆盖是指子类对从父类继承过来的数形变量及方法可以重新加以定义。

◆ 属性的隐藏

子类中声明了雨超类相同的成员变量名。隐藏了之后依然存在。子类中拥有了两个相同名字的变量，一个继承自超类，另一个由自己声明。

例如：

```
class Parent{
    Number aNumber;
}
class Child extends Parent{
    Float aNumber;//成员变量名一样，但是类型不一样，这样是将超类的变量名隐藏了
}
```

当子类执行继承自超类的操作时，处理的是继承自超类的变量，而当子类执行它自己声明的方法时，所操作的就是它自己声明的变量。

那么如何访问被隐藏的超类属性呢？

（1）调用从超类继承的方法，则操作的是从超类继承的属性；

（2）本类中声明的方法使用“super.属性”访问从超类继承的属性。

具体可以参看课本上的例子。

静态成员不可以被继承，但可以被所有的子类访问。

图 1

关于图 1 的解释：

要看出来，在这里的 class A 中，static 是一个静态成员，它不可以被子类继承但是可以被子类访问。因此，我们在执行完 a1.setx(4)之后，x 的值被修改为了 4，输出结果为 4.

接下来对子类 B 进行操作。在 static 后，整个内存里 x 就是这一个，所以此时调用出来的 x 也是 x=4，而不是在不用 static 时候的结果（那个时候 x=2）。所以此时无论 super.x 还是再调用 printa()方法结果都为 14。

同理，利用 b1.setx(6)之后，x 就设置为了 6。

用不用 static 的差别就在于：用了 static，整个内存空间就一个 x，无论对超类还是子类操作，x 都是同一个，因此只要修改了就是修改了；不用 static，子类就相当于复制了一个超类的 x，子类修改了父类不一定修改。

◆ 方法覆盖

如果子类不需要使用超类继承过来的方法的功能，则可以声明自己的同名方法，称为方法覆盖。

注意：

覆盖方法的返回类型、方法名称、参数的个数和类型必须和被覆盖的方法一模一样。

只需要在方法名前使用不同的类名或不同类的对象名即可区分覆盖方法和被覆盖方法。

覆盖方法的访问权限可以比被覆盖的更宽松，但是不能更为严格。例如超类中是 public，子类必须是 public；如果超类是 private，子类可以是 protected 或者 public。

应用场合：

- 子类实现的超类相同的功能，但采用不同的算法或公式；
- 在名字相同的方法中，要做比超类更多的事情；
- 取消从超类继承的方法。

注意事项：

（一）必须覆盖的方法：

（1）派生类必须覆盖基类中的抽象方法，否则派生类自身也称为抽象类

（二）不能覆盖的方法：

（1）基类中的 final 的终结方法；

（2）声明 static 的静态方法。

在 JAVA 中，使用构造方法是生成实例对象的唯一方法，它可以重载，但不能从父类那里继承，它的名字必须与它所在的类名字完全相同，并且不返回任何数据类型也不能是 void 类型。

有继承时构造方法遵循以下原则：

- 子类不能从父类继承构造方法。
- 好的程序设计方法是在子类的构造方法中调用某一个父类构造方法。
- super 关键字也可以用于构造方法中，其功能为调用父类的构造方法。
- 如果在子类的构造方法的声明中没有明确调用父类的构造方法，则系统在执行子类的构造方法会自动调用父类的默认构造方法（即无参的构造方法）。
- 如果在子类的构造方法的声明中调用父类的构造方法，则调用语句必须出现在子类构造方法的第一行。

3.2 OBJECT 类

OBJCET 类是 java 所有类的直接或间接父类，也是类库中所有类的父类，处在类层次最高点。所有的其它类都是从 object 类派生出来的。

图 2

◆ equals()方法

相等和同一

相等（equal）是两个对象具有相同的类型，及相同的属性值。

如果两个引用变量指向同一个对象，则称这两个引用变量同一（identical）。

同一，一定相等；相等不一定同一。

比较运算符“==”在基本运算用来比较相等。当用来判断对象的引用的时候，比较这两个引用如果是相等，那么同一。

如果只想用 equals()判断是否相等，需要覆盖方法体，对其进行重写。定义代码如下：

```
//这只是一个示范的例子需要根据具体情况再定。
public boolean equals(Object obj){
    if(obj instanceof Apple){
```

```
            Apple a = (Apple)obj;
            return(color.equals(a.getColor())&&(ripe == a.getRipe()));
        }
        return false;
}
```

◆ hashCode()方法

作用：返回对象散列码的方法，该方法实现的一般规定是：

在一个 Java 程序的一次执行过程中，如果对象“相等比较”所使用的信息没有被修改的话，同一对象执行 hashCode 方法每次都应返回同一个整数，在不同的执行中，对象的 hashCode 方法返回值不必一致。

如果依照 equals 方法两个对象是相等的，则在这两个对象上调用 hashCode 方法应该返回同样的整数结果；如果不相等，并不要求返回值不同。

只要实现合理，object 类定义的 hashCode 方法为不同对象返回不同的整数。一个典型的实现是，将对象的内部地址转换成整数返回，但是 java 并不要求必须这样实现。

◆ clone()方法

使用clone方法复制对象，必须覆盖clone方法，因为在Object类中其被定义为protected，所以要覆盖为 public。需要实现 Cloneable 接口，作为一个标记，赋予一个对象呗克隆的能力。

◆ finalize()方法

在对象被垃圾回收器回首之前，系统自动调用对象的 finalize()方法。

如果要覆盖 finalize 方法，覆盖方法的最后必须调用 super.finalize。

◆ getClass()方法

它是一个 final 方法，返回一个 Class 对象，用来代表对象所属的类。

通过 Class 对象，可以查询各种信息，如名字、超类、实现接口的名字等。

3.3 终结类与终结方法

是用 final 修饰的类和方法；

终结类不能被继承，终结方法不能被子类覆盖。

◆ **final 类举例**

```
//声明 ChessAlgorithm 类为 finall 类
final class ChessAlgorithm{
    //代码
}
//如果编写如下程序:
class BetterChessAlgorithm extends ChessAlgorithm{
    //代码
}
```

则编译器会出现报错。

◆ **final 方法举例**

```
class Parent{
    public Parent(){}//构造方法
    final int getPI(){return Math.PI;}//终结方法
}
```

getPI()在这用 final 修饰符声明是终结方法，不能在子类中对该方法覆盖，因而如下声明是错误的：

```
class Child extends Parent{
    public Child(){}
    int getPI(){return 3.14;}//覆盖会报错
}
```

3.4 抽象类

3.4.1 抽象类的声明

- 类名前加修饰符 abstract；
- 可以包含常规类能包含的任何成员，包括非抽象方法；
- 也可以包含抽象方法：用 abstract 修饰，只有方法原型，没有方法实现；
- 没有具体实例对象的类，不能使用 new 方法进行实例化，只规定了一些接口，只能用作超类；
- 只有当子类实现了抽象超类中所有的抽象方法，子类才不是抽象类，才能产生实例；
- 子类中仍有未实现的抽象方法，则该子类也是抽象类，那么也不能产生实例。

抽象类声明语法形式：

```
abstract class Number{…}
```

3.4.2 抽象方法

抽象方法归并了一种行为的访问接口，通常放在抽象类中规定所有的子类必须得有这样的行为，但是这个行为的具体实现、具体算法等无法实现，所以将它携程抽象方法，用 abstract 来修饰。

语法形式：

```
public abstract <returnType><methodName>(…);
```

说明：

- 仅有方法原型，没有方法体；
- 抽象方法的具体实现由子类在它们各自的类声明中完成；
- 只有抽象类可以包含抽象方法。

抽象方法的优点：

- 隐藏具体的细节信息，使调用该方法的程序不必过分关注该类和它子类的内部状况。所有的子类使用的都是相同的方法原型，其中包含了调用该方法时需要了解的全部信息；
- 强迫子类完成指定的行为，规定所有子类的“标准”行为。

3.5 泛型

泛型的本质是将类型参数化。包括泛型类、泛型方法、泛型接口。泛型接口在下一讲介绍。

泛型类示例：

```
class GeneralType <Type>{
    Type object;
    public GeneralType(Type object){
        this.object = object;
    }
    public Type getObj(){
        return object;
    }
}
public class Test{
    public static void main(String args[]){
        GeneralType<Integer> i = new GeneralType<Integer>(2);
        GeneralType<double> d = new GeneralType<Double>(0.33);
        System.out.println("i.object= " + (Integer)i.getObj());
        System.out.println("i.object= " + (Integer)d.getObj());//编译错误，d 指向的对象是
double，所以传递过来的是 double，但是此时强硬转成 integer 是不行的
    }
}
```

泛型方法示例：

```
class GeneralMethod{
    <Type> void printClassName(Type object){
        System.out.println(object.getClass().getName());
    }
}
public class Test{
    public static void main(String[] args){
        GeneralMethod gm = new GeneralMethod();
        gm.printClassName("Hello");
        gm.printClassName(3);
        gm.printClassName(3.0f);
        gm.printClassName(3.0);
    }
```

使用通配符（即“?”），就能使这个程序更为通用。举例如下：

```
class GeneralType <Type>{
    Type object;
    public GeneralType(Type object){
        this.object = object;
    }
    public Type getObj(){
        return object;
    }
}
```

```
class ShowType{
//给它不同类型实参构造的对象
    public void show(GeneralType<?> o) {
        System.out.println(o.getObj().getClass().getName());
    }
}
public class Test{
    public static void main(String args[]){
        ShowType st = new ShowType();
        GeneralType<Integer> i =new GeneralType<Integer> (2);
        GeneralType<String> s =new GeneralType<String> ("Hellow");
        st.show(i);
        st.show(s);
    }
}
```

程序运行结果：

```
java.lang.Integer
java.lang.String
```

有限制的泛型：

在参数“Type”后面使用 extends 关键字并加上类名或者接口名，表明参数所代表的类型必须是该类的子类或者实现了该接口。

注意，对于实现了某接口的有限制泛型，也是使用 extends 关键字而不是 implements 关键字。

举例：

```
class GeneralType <Type extends Number>{
    Type object;
    public GeneralType(Type object){
        this.object = object;
    }
    public Type getObj(){
        return object;
    }
}

public class Test{
    public static void main(String args[]){
        ShowType st = new ShowType();
        GeneralType<Integer> i =new GeneralType<Integer> (2);
        //GeneralType<String> s =new GeneralType<String> ("Hellow");
        //非法，T 只能是 Number 或者 Number 的子类
    }
}
```

3.6 类的组合

我们设计新的类的时候可以选择已有类的对象作为新类的成员，这就构成了一个组合机制。

类的组合也是一种类的重用机制，表达的是包含关系。

组合的语法：

（1）将已经存在的类的对象放到新类中。例如，可以说“厨房里有一个路子和一个冰箱”。所以，课简单的把对象 myCooker 和 myRefrigerator 放在类 kitchen 中：

```
class Cooker{//类的语句}
class Refrigerator{//类的语句}
class Kitchen{
   Cooker myCooker;
   Refrigerator myRefrigerator;
}
```

组合距离-线段类：

```
public class Point //点类
{
      private int x,y;//coordinate
      public Point(int x, int y){this.x = x; this.y =y;}
      public int GetX(){return x;}
      public int GetY(){return y;}
}
class Line{
      private Point p1,p2;
      Line(Point a, Point b){
            p1 = new Point(a.GetX(),a.GetY());
            p2 = new Point(b,GetX(),b.GetY());
      }
      public double Length(){
            return Math.sqrt(Math.pow(p2.GetX()-p1.GetX(),2)
            +Math.pow(p2.GetY()-p1.GetY(),2));
      }
}
```

继承表达的是隶属关系；组合表达的是包含关系。