Java语言程序设计（第2版）

第 4 章　类的重用

郑　莉

导学

目录

类的继承

- 如何充分重用已有的设计和实现，在已有类的基础上设计新的类？
- 继承已有类的全部属性和行为，在此基础上改造、扩展

Object类

- 所有Java类的直接或间接超类。
- 如果一个类没有继承任何超类，那么就隐含直接继承了Object类

终结类与终结方法

- 有些类，或类的方法从安全的角度或者算法的角度不希望被修改，就可以设为终结类、终结方法
- 终结类不可以被继承（扩展）
- 终结方法不可以被覆盖

抽象类

- 如果希望统一规定整个类家族的共同属性和行为，但是又没有统一抽象的行为实现，就可以声明一个抽象超类，包括所有需要子类具有的属性和实现的方法，在超类中无法实现的方法就声明为抽象方法，留待子类各自去实现。
- 由于抽象类中允许有未实现的方法，因此不能生成对象，只能当做超类使用，主要是为了设计的目的和隐藏实现细节

泛型

- 不仅数据可以作为参数，类型也可以作为参数。
- 使用类型参数，可以使程序更为通用，适用于处理不同类型的数据
- 这就是泛型

类的组合

- 组合是另一种类重用机制——部件组装的思想
- 在声明一个新的类时，用已有类的对象作为成员

这就是第3章将要学习的主要内容

类继承的概念和语法

<3.1.1>、<3.1.2>

类继承的概念

- 一种由已有的类创建新类的机制，是面向对象程序设计的基石之一

类继承的概念

- 根据已有类来定义新类，新类拥有已有类的所有功能
- Java只支持类的单继承，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的集合，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

超类和子类

- 超类(superclass)
  - 也称基类(base class)
  - 是被直接或间接继承的类
- 子类 (subclass)
  - 也称派生类(derived-class)
  - 继承其他类而得到的类
  - 继承所有祖先的状态和行为
  - 派生类可以增加变量和方法
  - 派生类也可以覆盖(override)继承的方法

超类和子类

- 子类对象与超类对象存在“是一个.....”(或“是一种.....”)的关系

子类对象

- 从外部来看，它应该包括
  - 与超类相同的接口
  - 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

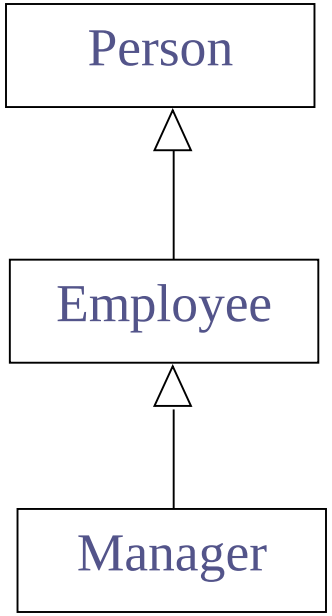
继承的语法

继承的语法

```
[ClassModifier] class ClassName extends SuperClassName
{
//类体
}
```

类继承举例

- 设有三个类：Person, Employee, Manager。其类层次如图：



例：类继承

```
public class Person {
  public String name;
  public String getName() {
   return name;
  }
}

public class Employee extends Person {
    public int employeeNumber;
    public int getEmployeeNumber() {
       return employeeNumber;
    }
}

public class Manager extends Employee {
    public String responsibilities;
    public String getResponsibilities() {
      return responsibilities;
    }
}
```

例：类继承

```
public class Exam4_2Test {
  public static void main(String args[]){
    Employee li = new Employee();
    li.name = "Li Ming";
    li.employeeNumber = 123456;
    System.out.println(li.getName());
    System.out.println(li.getEmployeeNumber());

    Manager he = new Manager();
    he.name = "He Xia";
    he.employeeNumber = 543469;
    he.responsibilities = "Internet project";
    System.out.println(he.getName());
    System.out.println(he.getEmployeeNumber());
    System.out.println(he.getResponsibilities());
  }
}
```

运行结果

```
Li Ming
123456
He Xia
543469
Internet project
```

例：访问从超类继承的成员

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

```
public class B {
    public int a = 10;
    private int b = 20;
    protected int c = 30;
    public int getB()  { return b; }
}

public class A extends B {
    public int d;
    public void tryVariables() {
        System.out.println(a);          //允许
        System.out.println(b);          //不允许
        System.out.println(getB());     //允许
        System.out.println(c);          //允许
    }
}
```

<人像>——结束语

这一节我们学习了类继承的概念、语法，并通过简单的例题观察了如何从一个已有的类派生出辛德雷。

隐藏和覆盖

< 3.1.3 >

子类对从超类继承来的属性变量及方法可以重新定义

属性的隐藏

属性的隐藏

- 子类中声明了与超类中相同的成员变量名
  - 从超类继承的变量将被隐藏
  - 子类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
  - 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

```
class Parent {
    Number aNumber;
}
```

```
class Child extends Parent {
    Float aNumber;
}
```

访问被隐藏的超类属性

- 调用从超类继承的方法，则操作的是从超类继承的属性
- 本类中声明的方法使用“super.属性”访问从超类继承的属性

例：属性的隐藏

```
class A1
{   int x = 2;
    public void setx(int i)
    {   x = i;    }
    void printa()
    {System.out.println(x);}
}
class B1 extends A1
{   int x=100;
    void printb()
    {  super.x = super.x +10 ;
       System.out.println("super.x=" +
                    super.x + "   x= " + x);
    }
}
```

```
public class Exam4_4Test
{  public static void main(String[] args)
    {   A1 a1 = new A1();
        a1.setx(4);
        a1.printa();

        B1 b1 = new B1();
        b1.printb();
        b1.printa();

        b1.setx(6);  // 将继承x值设置为6
        b1.printb();
        b1.printa();
        a1.printa();
    }
}
```

运行结果：

```
4
super.x= 12  x= 100
12
super.x= 16  x= 100
16
4
```

例：访问超类静态属性

```
class A {
    static int x = 2;
    public void setx(int i) {x = i;}
    void printa() {
        System.out.println(x);
    }
}
class B extends A {
    int x = 100;
    void printb() {
        super.x = super.x + 10;
        System.out.println("super.x= " + super.x + "
    x= " + x);
    }
}
```

```
public class Tester {
    public static void main(String[] args)
{
        A a1 = new A();
        a1.setx(4);
        a1.printa();
        B b1 = new B();
        b1.printb();
        b1.printa();
        b1.setx(6);
        b1.printb();
        b1.printa();
        a1.printa();
    }
}
```

运行结果：

```
4
super.x= 14  x= 100
14
super.x= 16  x= 100
16
16
```

方法覆盖

方法覆盖

- 如果子类不需使用从超类继承来的方法的功能，则可以声明自己的同名方法，称为方法覆盖
- 覆盖方法的返回类型，方法名称，参数的个数及类型必须和被覆盖的方法一摸一样
- 只需在方法名前面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宽松，但是不能更为严格

方法覆盖的应用场合

- 子类中实现与超类相同的功能，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；
- 在子类中需要取消从超类继承的方法。

方法覆盖的注意事项

- 必须覆盖的方法
  - 派生类必须覆盖基类中的抽象的方法，否则派生类自身也成为抽象类.
- 不能覆盖的方法
  - 基类中声明为final的终结方法
  - 基类中声明为static 的静态方法
- 调用被覆盖的方法
  - super.overriddenMethodName();

<人像>——结束语

这一节我们学习了在子类中如何隐藏从超类继承的方法，如何覆盖从超类继承的成员。这样就可以在继承的基础上进行修改

子类的构造方法

<3.1.4>

有继承时子类的构造方法遵循以下的原则

- 子类不能从超类继承构造方法
- 最好在子类的构造方法中使用super关键字显式调用超类的某个构造方法，调用语句必须出现在子类构造方法的第一行。
- 如子类构造方法体中没有显式调用超类构造方法，则系统在执行子类的构造方法时会自动调用超类的默认构造方法（即无参的构造方法）

例：子类的构造方法

```
public class Person {
    protected String name, phoneNumber, address;
    public Person() {
        this("", "", "");
    }
    public Person(String aName, String aPhoneNumber, String anAddress) {
        name = aName;
        phoneNumber = aPhoneNumber;
        address = anAddress;
    }
}
```

例：子类的构造方法（续）

```
public class Employee extends Person {
    protected int employeeNumber;
    protected String workPhoneNumber;
    public Employee() {
        //此处隐含调用构造方法 Person()
        this(0, "");
    }
    public Employee(int aNumber, String aPhoneNumber) {
        //此处隐含调用构造方法 Person()
        employeeNumber = aNumber;
        workPhoneNumber = aPhoneNumber;
    }
}
```

例：子类的构造方法（续）

```
public class Professor extends Employee {
    protected String research;
    public Professor() {
        super();
        research = "";
    }
    public Professor(int aNumber, String aPhoneNumber, String aResearch)
     {
        super(aNumber, aPhoneNumber);
        research = aResearch;
    }
}
```

这一节介绍了在有继承的情况下，子类的构造方法怎么写。

应该首先在第一句调用超类的构造方法

如果不显式地调用超类的构造方法，超类的默认构造方法将被调用。

Object 类

<3.2>

Object 类

- 所有类的直接或间接超类，处在类层次最高点；
- 包含了所有Java类的公共属性。

Object类的主要方法

Object类的主要方法

- public final Class getClass()
  - 获取当前对象所属的类信息，返回Class对象。
- public String toString()
  - 返回表示当前对象本身有关信息的字符串对象。
- public boolean equals(Object obj)
  - 比较两个对象引用是否指向同一对象，是则返回true，否则返回false。
- protected Object clone( )
  - 复制当前对象，并返回这个副本。
- Public int hashCode()
  - 返回该对象的哈希代码值。
- protected void finalize() throws Throwable
  - 在对象被回收时执行，通常完成的资源释放工作。

相等和同一

- 在Object类中声明的equals()方法功能是比较两个对象引用是否指向同一对象，而不是比较两个引用指向的对象是否相等。
- 接下来我们看一下“相等”和“同一”的区别。

相等和同一

- 两个对象具有相同的类型，及相同的属性值，则称二者相等(equal)。
- 如果两个引用变量指向的是同一个对象，则称这两个引用变量同一(identical)。
- 两个对象同一，则肯定相等。
- 两个对象相等，不一定同一。
- 比较运算符“==”判断的是这两个对象是否同一。

例：用“==”判断两个引用是否同一

```
public class Tester1{
    public static void main(String args[]) {
        BankAccount a = new BankAccount("Bob", 123456, 100.00f);
        BankAccount b = new BankAccount("Bob", 123456, 100.00f);
        if (a == b) {
            System.out.println("YES");
        } else {
            System.out.println("NO");
        }
    }
}
```

- BankAccount类在2.5.1中声明，此程序运行的结果为“NO”，原因是使用“==”判断的是两个引用是否同一。

例：用“==”判断两个引用是否同一（续）

```
public class Tester2 {
    public static void main(String args[]) {
        BankAccount a = new BankAccount("Bob", 123456, 100.00f, Grade.General);
        BankAccount b = a;
        if (a == b) {
            System.out.println("YES");
        } else {
            System.out.println("NO");
        }
    }
}
```

- a与b指向的是同一个对象，a与b同一。输出结果为“YES”

equals 方法

- 由于Object是类层次结构中的树根节点，因此所有其他类都继承了equals()方法
- equals()方法是像方法名表达的意思那样，判断两个对象相等吗？

equals 方法

- Object类中的 equals() 方法的定义如下：

```
public boolean equals(Object x) {
    return this == x;
}
```

- 也是判断两个对象引用是否同一。

例：equals方法

```
public class EqualsTest{
        public static void main(String args[]){
        BankAccount a = new BankAccount("Bob", 123456, 100.00f);
        BankAccount b = new BankAccount("Bob", 123456, 100.00f);
       if (a.equals(b))
          System.out.println("YES");
       else
          System.out.println("NO");
    }
}
```

- 由于两个引用不是指向同一对象，运行结果仍然是“NO”

覆盖equlas方法

- 要判断两个对象各个属性域的值是否相同，则不能使用从Object类继承来的equals方法，而需要在类声明中对equals方法进行覆盖。
- 例如：String类中已经覆盖了Object类的equals方法，可以判别两个字符串是否内容相等。
- 方法原型必须与Object类中的equals方法完全相同。

例：覆盖equlas方法（1）

- 在BankAccount类中覆盖equals方法

```
public boolean equals(Object x) {
    if (this.getClass() != x.getClass())
            return false;
    BankAccount b = (BankAccount) x;
    return
      ((this.getOwnerName().equals(b.getOwnerName()))
      &&(this.getAccountNumber() == b.getAccountNumber())
      &&(this.getBalance() == b.getBalance()));
}
```

例：覆盖equlas方法（2）

```
public class Apple {
  private String color;
  private boolean ripe;
  public Apple(String aColor, boolean isRipe) {
     color = aColor;
     ripe = isRipe;
  }
  public void setColor(String aColor) { color = aColor;  }
  public void setRipe(boolean isRipe) { ripe = isRipe; }
  public String getColor() { return color; }
  public boolean getRipe() { return ripe; }
  public String toString() {
     if (ripe)  return("A ripe " + color + " apple");
     else  return("A not so ripe " + color + " apple");
  }
```

例：覆盖equlas方法（2）

```
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

- hashCode是一个返回对象散列码的方法

hashCode方法

- hashCode是一个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。
  - 如果依据equals方法两个对象不相等，并不要求在这两个对象上调用hashCode方法返回值不同。
- 只要实现得合理，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为整数返回，但是Java语言并不要求必须这样实现。

clone方法

- 用于根据已存在的对象构造一个新的对象，也就是复制对象。

clone方法

- 使用clone方法复制对象
  - 覆盖clone方法：在Object 类中被定义为protected，所以需要覆盖为public。
  - 实现Cloneable 接口，赋予一个对象被克隆的能力(cloneability)

```
class MyObject implements Cloneable
{ //…
}
```

finalize方法

- 在对象被垃圾回收器回收之前，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

- 通过getClass方法可以获得一个对象所属类的信息

getClass方法

- 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，可以查询类的各种信息：比如名字、超类、实现接口的名字等。
- 例如：

```
void PrintClassName(Object obj) {
   System.out.println("The Object's class is " + obj.getClass().getName());
}
```

notify、notifyAll、wait方法

- final方法，不能覆盖
- 这三个方法主要用在多线程程序中

本节介绍了Object类以及其中的常用方法。一些方法需要子类覆盖，形成每个子类的特殊实现
终结方法是不可覆盖的，下一节介绍

终结类与终结方法

<3.3>

终结类与终结方法

- 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。

例：终结类

- 声明ChessAlgorithm 类为final 类

  final class ChessAlgorithm { . . . }

- 如果写下如下程序：

  class BetterChessAlgorithm extends ChessAlgorithm { … }

  编译器将显示一个错误

  ```
  Chess.java:6: Can't subclass final classes: class ChessAlgorithm
  class BetterChessAlgorithm extends ChessAlgorithm {
        ^
  1 error
  ```

例：终结方法

- final 方法举例

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

- 说明

getPI()是用final修饰符声明的终结方法，不能在子类中对该方法进行覆盖，因而如下声明是错的：

```
Class Child extends Parent
{
   public Child() {}
   int getPI() { return 3.14; } //错！不允许覆盖超类中的终结方法
}
```

- 这一节介绍了终结类和终结方法；
- 终结类不能被继承
- 终结方法不能覆盖

抽象类

<3.4>

抽象类

- 代表一个抽象概念的类；
- 规定整个类家族都必须具备的属性和行为。

抽象类

- 类名前加修饰符abstract；
- 可包含常规类能包含的任何成员，包括非抽象方法；
- 也可包含抽象方法：用abstract修饰，只有方法原型，没有方法的实现；
- 没有具体实例对象的类，不能使用new方法进行实例化，只能用作超类；
- 只有当子类实现了抽象超类中的所有抽象方法，子类才不是抽象类，才能产生实例；
- 如果子类中仍有抽象方法未实现，则子类也只能是抽象类。

抽象类声明的语法形式

```
abstract class Number {
    . . .
}
```

如果写：
new Number();
编译器将显示错误

抽象方法

- 规定子类应该具有的行为，但在抽象超类中尚不能实现的方法，可以声明为抽象方法。

抽象方法

- 声明的语法形式为：

  public abstract <returnType> <methodName>(...);
- 仅有方法原型，而没有方法体；
- 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。

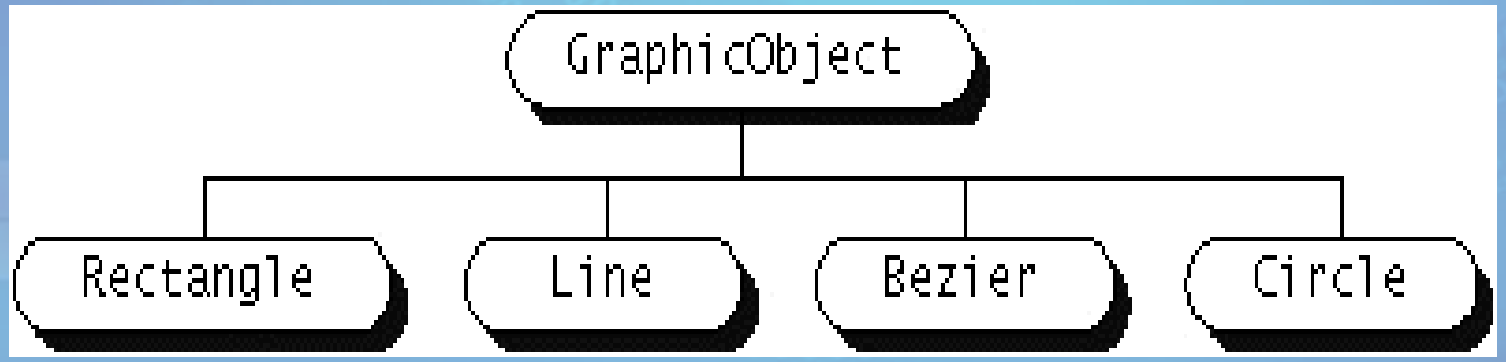
抽象方法的优点

- 隐藏具体的细节信息，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。

例：抽象的绘图类和抽象方法

- 各种图形都需要实现绘图方法，可在它们的抽象超类中声明一个draw抽象方法

```
abstract class GraphicObject {
    int x, y;
    void moveTo(int newX, int newY) { . . . }
    abstract void draw();
}
```



例：抽象的绘图类和抽象方法

- 然后在每一个子类中重写draw方法，例如：

```
class Circle extends GraphicObject {
    void draw() { . . . }
}
class Rectangle extends GraphicObject {
    void draw() { . . . }
}
```

泛型

<3.5>

泛 型

- 泛型的本质是参数化类型，即所操作的数据类型被指定为一个参数；
- 可以声明泛型类、泛型方法和泛型接口（下一章介绍接口）。

- 下面通过例题演示一下泛型类和泛型方法。

例：泛型类

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

例：泛型类

```
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
    }
}
```

例：泛型方法

```
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);
    }
}
```

通配符泛型

- 使用通配符可以使程序更为通用

例：使用通配符

```
class GeneralType <Type> {
    Type object;
    public GeneralType(Type object) {
        this.object = object;
    }
    public Type getObj() {
        return object;
    }
}
class ShowType {
    public void show (GeneralType<?> o) {
        System.out.println(o.getObj().getClass().getName());
    }
}
```

例：使用通配符

```
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> ("hello");
        st.show (i);
        st.show(s);
    }
}
```

程序的运行结果如下：

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

有限制的泛型

- 虽然可以使用泛型使程序能够处理多种不同类型对象，并不是所有对象都能进行某些操作；
- 有时候需要将泛型中参数代表的类型做限制，此时就可以使用有限制的泛型。

有限制的泛型

- 在参数“Type”后面使用“extends”关键字并加上类名或接口名，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

例：有限制的泛型

```
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[]){
        GeneralType<Integer> i = new GeneralType<Integer> (2);
        //GeneralType<String> s = new GeneralType<String> ("hello");
        //非法，T只能是Number或Number的子类
    }
}
```

类的组合

<3.6>

组合的意义

- 面向对象的程序用软件对象来模仿现实世界的对象：
  - 现实世界中，大多数对象由更小的对象组成；
  - 与现实世界的对象一样，软件中的对象也常常是由更小的对象组成。
- Java的类中可以有其他类的对象作为成员，这便是类的组合。
- 组合也是一种重用机制，可以使用“有一个”来描述这种关系。

组合的语法

- 将已存在类的对象放到新类中即可
- 例如，可以说“厨房（kitchen）里有一个炉子（cooker）和一个冰箱（refrigerator）”。所以，可简单的把对象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;
}
```

例：组合举例——线段类

- 一条线段包含两个端点

```
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; }
}
```

例：组合举例——线段类（续）

```
class Line   //线段类
{
   private Point  p1,p2;     // 两端点
   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));
    }
}
```

组合与继承的比较

- “包含”关系用组合来表达
- “属于”关系用继承来表达

组合与继承的结合

- 许多时候都要求将组合与继承两种技术结合起来使用，创建一个更复杂的类

例：组合与继承的结合

```
class Plate {  //声明盘子
        public Plate(int i) {
                System.out.println("Plate constructor");
        }
    }
    class DinnerPlate extends Plate { //声明餐盘为盘子的子类
        public DinnerPlate(int i) {
                super(i);
                System.out.println("DinnerPlate constructor");
        }
    }
```

例：组合与继承的结合（续）

```
class Utensil { //声明器具
    Utensil(int i) {
            System.out.println("Utensil constructor");
    }
}
class Spoon extends Utensil { //声明勺子为器具的子类
    public Spoon(int i) {
            super(i);
            System.out.println("Spoon constructor");
    }
}
```

例：组合与继承的结合（续）

```
class Fork extends Utensil { //声明餐叉为器具的子类
    public Fork(int i) {
            super(i);
            System.out.println("Fork constructor");
    }
}
class Knife extends Utensil { //声明餐刀为器具的子类
    public Knife(int i) {
            super(i);
            System.out.println("Knife constructor");
    }
}
```

例：组合与继承的结合（续）

```
class Custom { // 声明做某事的习惯
        public Custom(int i) {
            System.out.println("Custom constructor");
        }
}
```

例：组合与继承的结合（续）

```
public class PlaceSetting extends Custom {//餐桌的布置
    Spoon sp; Fork frk; Knife kn;
    DinnerPlate pl;
    public PlaceSetting(int i) {
        super(i + 1);
        sp = new Spoon(i + 2);
        frk = new Fork(i + 3);
        kn = new Knife(i + 4);
        pl = new DinnerPlate(i + 5);
        System.out.println("PlaceSetting constructor");
    }
    public static void main(String[] args) {
        PlaceSetting x = new PlaceSetting(9); }
}
```

例：组合与继承的结合（续）

- 运行结果

```
Custom constructor
Utensil constructor
Spoon constructor
Utensil constructor
Fork constructor
Utensil constructor
Knife constructor
Plate constructor
DinnerPlate constructor
PlaceSetting constructor
```

第3章小结

小结

- 介绍了Java语言类的重用机制，形式可以是继承或组合
- Object类的主要方法
- 终结类和终结方法的特点和语法
- 抽象类和抽象方法的特点和语法