第 5 章　Java 中的面向对象思想与技术

【本章内容提要】

本章介绍 Java 面向对象的三大特性，即封装、继承和多态。给出对象和类的基本概念、类的定义、构造函数、对象的创建和使用、this 关键字、static 关键字、final 关键字、类的继承、方法的重写、接口、访问控制等。这些面向对象的技术有效地提高程序员的工作效率，方便程序员在更大规模软件设计中的合作。Java 的面向对象知识具有内容清晰明了、结构层次分明和简单易用的特点。对于 Java 的面向对象知识的学习不仅需要掌握它的语法结构，更为重要的是深入体会面向对象设计思想。

【学习目标和要求】

1. 了解面向对象程序设计方法。
2. 掌握类的概念和定义方法。
3. 掌握对象的创建和简单使用。
4. 理解构造函数的特点。
5. 理解继承和多态的概念。
6. 理解继承中父类和子类对象的创建过程。
7. 初步掌握 this 和 super 的用法。
8. 了解常用修饰符的用法。
9. 掌握接口的定义和使用方法。

本章重点：类的概念、类的定义、对象创建。

本章难点：内存中对象的创建，继承中父类和子类对象的创建过程，访问控制修饰符的作用。

5.1　一个简单的类

通过一个简单的类引入 Java 的面向对象知识。

如图 5-1 所示，程序中声明了一个 Time 类，包含三个成员属性，通过构造方法 Time ( ) 为成语属性赋初值，并通过方法内的语句建立三个属性间的联系和限定。

【例 5-1】

```
import java.util.*;
public class Time{
        int hour;
        int minute;
        int second;

        Time(int h,int m,int s){
           if(h>=0&& h<23){
                hour=h;
           }else{
               System.out.println("the hour is error");
           }

           if(m>=0&& m<59){
                minute=m;
           }else{
                System.out.println("the minute is error");
           }

           if(s>=0&& s<59){
                second=s;
           }else{
                System.out.println("the second is error");
           }
        }
}
```

图 5-1　一个简单的类

面向过程的语言是建立在数学思维上的，因此把事物的属性看作变量，割断了事物属性之间的联系。例如面向规程语言定义一个时间为如下定义方式：

```
int second;
int minute;
int hour;
```

面向过程语言执行该语句时为三个变量分配独立的存储空间，三者之间没有内在联系。这种构造方法存在以下一些缺陷。

（1）如果要声明多个时间类型的变量，则声明的变量数量非常多，造成命名空间的浪费，而且也容易引发重名和管理混乱。

（2）没有有效的方式约束各个变量之间的关系，hour 的取值范围在 0 ~ 23 之间，minute 在 0~59 之间，second 在 0~59 之间。

此外，当 minute 超过 59 时，hour 增加 1，当 second 超过 59 时，minute 增加 1，显然程序中这些变量可以赋予比上述范围更大的值，当某个值变化需要引发其他值的逻辑变化也不能在程序中体现。因此通过过程语言来描述变量之间的关系存在一些缺陷。然而 Java 中，通过方法的设计可以对关联变量进行限制，也可以通过方法建立变量之间的动态变化关系，如图 5-1 中，在构造方法中对这些变量进行了限制和联系。

5.2 类定义和类成员

5.2.1 类定义

类是 Java 语言的核心内容和基本要素，类为对象提供模板，并封装了对象的方法和属性，因此在创建对象前必须创建对应的类。Java 中类定义的格式为

```
修饰符 class 类名{
  修饰符  类型  成员变量 l;
  修饰符  类型  成员变量 2;
……
  修饰符  类型  成员方法 1(参数列表){
    类型  局部变量;
      方法体
  }
  修饰符  类型  成员方法 2(参数列表){
    类型  局部变量;
      方法体
  }
……
}
```

类的定义以关键字 class 作为开始，其后为类名称，表明定义的是一个类。class 前的修饰符可以没有，也可以有多个用来限定类的使用方式。类名是用户为该类所起的名字，它的命名规则遵循标识符的命名规则。

类的内容包含在大括号内，包含有两部分内容，一部分是属性(数据成员变量)，可以含有任意多个，成员变量前面的类型是该变量的类型，修饰符限定属性的使用方式；另一部分是成员方法，也可以有任意多个，其前面的类型是方法返回值的类型，修饰符限定方法的使用方式，方法体是要执行的真正语句。在方法体中还可以定义该方法内使用的局部变量，这些变量的作用域限定在方法内有效。

关于类定义有如下几点说明。

(1) Java 中一个文件可以包含一个类或几个类，因此类的定义与实现是放在一起保存的；反之，一个类不能包含几个文件，编译后，一个 *.java 文件生成多个 *.class 文件，其数量与文件中的类数量相关。

(2) Java 源文件名必须根据文件中的公有类名来定义，即共有类名和文件名一致，并且要区分大小写，一般公有类名的第一个字母大写，公有类的修饰符为 public。

(3) 类定义中可以指明父类，也可以不指明。若没有指明从哪个类派生而来，则默认父类 Object。实际上，Object 是 Java 中所有类的父类。Java 中除 Object 之外的所有类均有一个且只有一个父类。Object 是唯一没有父类的类。

(4) class 定义的大括号之后没有分隔符“;”。

（5）对于类中的方法和属性的位置没有特殊的要求，一般分开集中书写，属性在上，而方法在下，或者反之都可以，用户可以根据自己的喜好选择一种。

（6）主方法 main( )可以存在多个类中，执行中根据命令后的类名，决定使用哪个主函数，即命令后的类中的 main( )方法被执行，默认执行时公有类中的主方法。

5.2.2　类的种类

Java 类分为两大类：系统给定类和自定义类。系统给定类由 JDK 给出，自定义类由程序员定义，本节主要介绍自定义类。自定义类必须是在程序中通过 class 关键字声明的类。

【例 5-2】如图 5-2 所示。

```
import java.util.Date;
public class Person{
        String name;
        Date birthday;

        void setBirthday(int year,int month,int date){
             birthday.setYear(year);
             birthday.setMonth(month);
             birthday.setDate(date);
        }
}
```

图 5-2　Person 类

如【例 5-2】所示，其中有两个类，Person 为自定义类，Date 为系统给定类。在 Date 类的对象 birthday 中包含有 year、month 和 date 三个类成员变量，Date 类的三个方法 setYear( )，setMonth( )，setDate( )分别给年、月、日赋值。掌握一定数量的系统类，不仅可以提高代码的书写效率，而且程序更为有效。

5.2.3　类的成员

类的成员也称为类的属性，组成类的数据单元。声明结构如下：

[修饰符]变量类型　变量名=[变量初始值];

修饰符分为访问控制修饰符、静态修饰符(static)和最终修饰符(final)。类成员变量类型既可以是简单数据类型，也可以是复杂数据结构类型，如数组和类。变量名满足标识符命名规则。

当分配对象内存时，Java 编译器自动初始化所分配的内存空间。数值变量赋初值 0。布尔变量赋初值 false。对于引用，即对象类型的任何变量，使用特殊值 null，注意 Java 中 null 应小写。在 Java 中，null 值表示引用不指向任何对象。运行过程中系统发现使用了这样一个引用时，程序立即停止进一步的访问，避免给系统带来任何危险。

自动初始化只用于成员变量，对方法中的自动变量不起作用。由于 Java 规定，任何变量使用之前，必须对变量赋值，所以方法内的自动变量不能进行自动初始化，因此要求程序员显式地对其赋值，当然可以给变量赋值为 null。如果在变量赋值之前使用它，编译器会指出一条错误信息。

类的成员变量作用域是整个类内，类内的成员变量不允许重名，其生命周期与对象的生命周期一致，当对象消亡时，该成员变量也被注销。

5.2.4 类的方法

在 Java 中，方法的定义的一般格式如下：

<修饰符><返回类型><名字><参数列表><块>

修饰符段可以含几个不同的修饰符，其中，限定权限的修饰符包括 public，protected 和 private。public 修饰符表示该方法可以被任何其他代码调用，而 private 表示方法只能被类中的其他方法调用。

名字是方法名，它必须使用合法的标识符。

返回类型说明方法返回值的类型。如果方法不返回任何值，它应该声明为 void，方法中 void 和 return 只能出现一个。Java 对待返回值的要求很严格，方法返回值必须与所说明的类型相匹配。如果方法说明有返回值，比如说是 int，那么方法从任何一个语句分支中返回时都必须返回一个整数值。

参数列表是传送给方法的参数表。表中各元素间以逗号分隔，每个元素由一个类型和一个标识符组成。

块表示方法体，是要实际执行的代码段。

5.2.5 类的成员方法 Get 和 Set

对类成员属性的使用在面向对象程序设计中不建议直接使用，因此类一般对成员属性进行限制，防止对数据的破坏。因此，通过方法操作类属性成为必然，获取属性值和赋值是类的属性最多的操作。

【例 5-3】Get 方法，如图 5-3 所示。

```
public class Person{
    String name;
    int age;
    int weight;
    int height;
    int ID;
    boolean sex;
    void setAge(int x){
        age=x;
    }
    void setName(String y){
        name=y;
    }
    int getAge(){
        return age;
    }
    String getName(){
        return name;
    }
}
```

图 5-3 Get 方法

【例 5-4】Set 方法，如图 5-4 所示。

```
public class Person{
    String name;
    int weight;
    int height;
    void getName(String n){
        name=n;
    }
    void getWeight(int w){
        weight=w;
    }
    void getheight(int h){
        height=h;
    }
    String getName(String n){
        return name;
    }
    int getWeight(int w){
        return weight;
    }
    void getheight(int h){
        return height;
    }
}
```

图 5-4　Set 方法

Java 的类程序设计中，一般不直接对成员变量进行直接操作，而是通过类方法来调用和修改成员变量的值，这类方法成为以 Get 和 Set 开头的类方法。使用这种方式有如下的优点。

(1)隐藏了 Java 内部成员变量，对成员的赋值和取值都是通过方法进行，切断直接操作成员变量。

(2)可以建立赋值过程中的约束，通过方法中的语句建立变量间的逻辑关系，避免对属性的非法操作，并提供提示信息。

(3)响应属性变化事件，由于方法中提供属性间的联系，因此一旦属性发生变化，就会激发方法中设定的事件。

5.3　对象的构造和初始化

类为对象提供定义，它规定对象内部的数据，创建该对象的特性以及对象的功能。因此对象创建晚于类创建，只有类定义后，该类的对象才可以存在。

5.3.1　对象创建

使用类创建的变量称为类类型变量，类类型变量的内存分配包括两部分。声明类变量时，在内存中为该变量建立一个引用，并置初值 null，表示不指向任何内存空间。然后程序中用操作符 new 申请相应的内存空间，内存空间基于 class 的定义，并将该段内存的首

址赋给刚才建立的引用。换句话说，用类类型说明的变量并不是数据本身，而只是对数据的引用(有时也称这样的引用为别名)，进一步要用 new 来创建类的实例。定义类以后，只有创建了该类的实例对象后才能使用该类。

对象引用，声明一个引用的格式如下：

类名　对象变量名；

例如可以声明 Person 类的变量：

Person p；

p 在没初始化之前，初值为 null，换句话说就是一个 Person 的对象并没有创建，只是给出一个能够指向 Person 对象的引用或别名。

一个类变量创建完成后，创建对象实例的格式如下：

对象变量名=new 类名([参数列表])；

实例化过程是为该对象分配内存。当一个对象实例不被任何变量引用时。Java 会自动启动垃圾回收线程回收它的内存空间。另外，当对象作为函数参数时，它传递的是对象引用，因此，方法内对参数的任何修改会影响到方法外。引用中实际存放的是对象地址，或者说是对象的句柄。

对象使用包括对对象成员和方法的使用，使用对象中的数据和方法的格式如下，需要使用操作符“.”。

对象引用．成员数据；

对象引用．成员方法(参数列表)；

例如，【例 5-5】是在前面定义的 Point 类基础上使用它的实例。

【例 5-5】 circles 类程序如图 5-5 所示。

```
import java.util.*;
class circles{
      public double r;
      double area;
      double girth;

      double compute_area(){
        area=r*r*3.1415;
        return area;
      }

      double compute_girth(){
        girth=2*r*3.1415;
        return girth;
      }

      public static void main(String[] args){
        circles c=new circles();
        c.r=10.0;
        System.out.println("the area is: "+c.compute_area());
        System.out.println("the girth is: "+c.compute_girth());

      }
}
```

图 5-5　circles 类程序

运行结果如图 5-6 所示。

```
.0_05 <Default> - <Default>--------------------
the area is: 314.15000000000003
the girth is: 62.830000000000005

Process completed.
```

图 5-6　circles 类程序运行结果

如图 5-5 所示，程序中建立 circle 类，在 main 方法中第一行使用操作符 new 建立一个 circle 类对象，在这里类类型变量声明和对象空间分配在一条语句中；第二行中通过操作符“.”给对象 c 中的成员变量半径 r 赋值；第三行和第四行调用成员方法计算周长和面积，并输出。

5.3.2　引用变最的赋值

Java 把说明为 class 类型的变量看作是引用，由此赋值语句的含义有了重大改变。引用之间的赋值不像简单变量之间的赋值。对变量 fname 和 lname，只存在一个 String 对象，它含有文本“Tom”，s 和 t 指向同一个对象，任何一个变量的修改，都会影响到另一个变量的值，如图 5-7 所示，参看【例 5-6】。

【例 5-6】引用变量赋值如图 5-7 所示。

```
class Test{
    public static void main(String[] args){
        String fname="Tom";
        String lname=fname;
        System.out.println(fname);
        System.out.println(lname);
        lname="John";
        System.out.println(fname);
        System.out.println(lname);
    }
}
```

图 5-7　引用变量赋值

运行结果如图所示 5-8 所示。

```
Tom
Tom
Tom
John

Process completed.
```

图5-8　引用变量赋值结果

5.3.3　对象初始化

前面已经讲过，在说明了引用后，要调用 new 为新对象分配空间。在调用 new 时，既可以带有变量，也可以不带变量。例如，在程序中可以写 new Button(“Press me”)。系统

根据所带参数的个数和类型，调用相应的构造方法。调用构造方法时，步骤如下。

（1）分配新对象的空间，并进行默认的初始化。在 Java 中，这个过程是不可分的，从而可确保不会有没有初值的对象。

（2）执行显式的成员初始化。

（3）执行构造方法。构造方法是一个特殊的方法。

如果在成员说明中写有简单的赋值表达式，就可以在构造对象时进行显式的成员初始化，如图 5-9 所示。

```
import java.util.*;
class circles{
        public double r=10;
        double area=r*r*3.1415;
        double girth=2*r*3.1415;

}
```

图 5-9　成员初始化

5.3.4　构造方法

构造方法是面向对象语言中一种特殊的方法，类对象的建立离不开构造方法。

【例 5-7】构造函数的方法如图 5-10 所示。

```
class Test{
    int a;
    double b;
    boolean c;
    char d;
    Test(){

    }
    Test(int x){
        a=x;
    }
    Test(int x,double y){
        a=x;
        b=y;
    }
    Test(int x,double y,boolean z){
        a=x;
        b=y;
        c=z;
    }
    Test(int x,double y,boolean z,char h){
        a=x;
        b=y;
        c=z;
        d=h;
    }
    public static void main(String[] args){
        Test t1=new Test();
        Test t2=new Test(10);
        Test t3=new Test(10,20.0);
        Test t4=new Test(10,20.0,true);
        Test t5=new Test(10,20.0,true,'char');
    }
}
```

图 5-10　构造函数的方法

如果需要在对象创建时对成员变量初始化，需要构造方法。构造方法由系统定义，也允许程序员编写自己的构造方法。构造方法是特殊的类方法，有着特殊的功能。它的名字与类名相同，没有返回值，在创建对象实例时由 new 运算符自动调用。同时为了创建实例的方便，一个类可以有多个具有不同参数列表的构造方法，且构造方法可以重载。一般情况下，不论系统给定类还是自定义类都具有多个构造方法，因此对系统类的学习不仅要记住类名称，而且要掌握它们构造方法的形式。

如图 5-10 所示，在类 Test 中定义了五个构造方法，其中一个的参数表是空的，另四个带有参数。在创建 Test 的实例时，可以使用上述的五种形式。因为构造方法的特殊性，它不允许程序员按通常调用方法的方式来调用。构造方法中参数列表的说明方式就决定了该类实例的创建方式。例如在 Test 中，使用语句 Test t=new Test(1，1)视为非法，因为类中没有 Test(int，int)构造方法。同时构造方法前不能有限定词如 native，abstract，synchronized 或 final，也不能从父类继承构造方法，因此构造方法不能继承得到。

Java 提供默认构造方法机制，每个类都至少有一个构造方法。如果程序员没有为类定义构造方法，系统会自动为该类生成一个默认的构造方法。默认构造方法的参数列表及方法体均为空，即如果 Test 没有构造方法，系统给定 Test( ){ }作为它的默认构造方法。如果程序员定义了一个或多个构造方法，则自动屏蔽掉默认构造方法。

类对象建立和构造方法的调用是同时进行的，每个对象建立只执行一次构造方法，类对象不能显示调用构造方法，但是类内构造方法之间可以互相调用。

默认构造方法的参数列表是空的，在程序中可以使用 new classname( )来创建对象实例，classname 是类名。如果程序员定义了构造方法，那么，最好包含一个参数表为空的构造方法，否则，调用 new classname( )时会出现编译错误。

5.3.5　析构方法

finalize 方法属于类，它可被所有类使用。如果对象实例不被任何变量引用时，Java 会自动进行“垃圾回收”，收回该实例所占用的内存空间。在对对象实例进行垃圾收集之前，Java 自动调用对象的 finalize 方法，用来释放对象所占用的系统资源。

finalize 方法的说明方式如下：

```
protected void finalize( ) throws Throwable
```

finalize 方法是 Object 类的成员方法，因此所有类都继承得到该方法，如果根据上述格式自定义 finalize 方法，则继承得到的 finalize 方法回被屏蔽。

构造方法和析构方法代表了一个对象的创建和消亡，它们都由系统自动地调用，构造方法是对象最先执行的方法，而析构方法是对象最后执行的方法。

【例 5-8】构造方法和析构方法示例如图 5-11 所示。

```
class Aobject{
    Aobject() {
        System.out.println("the object of Aobject is live!");
    }
    protected void finalize() throws Throwable{
        System.out.println("the object of Aobject is dead!");
    }
}
public class Bobject {
    Bobject() {
        System.out.println("the object of Bobject is live!");
    }
    protected void finalize() throws Throwable{
        System.out.println("the object of Bobject is dead!");
    }
    public static void main (String[] args) {
        for(int i=0;i<20;i++){
        Aobject A=new Aobject();
        Bobject B=new Bobject();}
        System.gc();
    }
}
```

图 5-11　析构方法程序

运行结果如图 5-12 所示。

```
the object of Aobject is live!
the object of Bobject is live!
the object of Aobject is live!
the object of Bobject is live!
the object of Aobject is live!
the object of Bobject is live!
the object of Aobject is live!
the object of Bobject is live!
the object of Aobject is live!
the object of Bobject is live!
the object of Bobject is dead!
the object of Aobject is dead!
the object of Bobject is dead!
the object of Aobject is dead!
the object of Bobject is dead!
the object of Aobject is dead!
the object of Bobject is dead!
the object of Aobject is dead!
the object of Bobject is dead!
the object of Aobject is dead!
the object of Bobject is dead!
```

图 5-12　程序结果

程序中每个对象创建时都会自动地调用构造方法，输出 the object of Aobject is live！语句和 the object of Bobject is live！语句。而当引用名指向另一个对象时，系统会通过方法 system. ge( )回收内存空间，将没有引用的对象注销，这时对象使用 finalize 方法释放空间，输出 the object of Aobject is dead！和 the object of Bobject is dead！语句。

5.4　继　　承

继承这一术语出现在不同的学科。法学中继承代表财产的继承，医学中父辈遗传给孩子的基因，而在面向对象程序设计里继承代表程序代码的复用，体现出一种资源传递的思想。在 Java 中，继承者称为子类，被继承者称为父类。

5.4.1　子类

在程序设计中，有时要建立关于某对象的模型，比如说 person，然后从这个最初的模型派生出多个具体化的版本，如学生 student，教师 teacher，工人 worker。显然，学生 student、教师 teacher 和工人 worker 都是人，他们都具有人的属性，除此之外，学生 student、教师 teacher 和工人 worker 都有各自的属性和方法。学生有自己的成绩，教师有自己的教授科目，工人有自己的专业技能等等。

【例 5-9】四个类的创建如图 5-13 所示。

```
class Person{
        String name;
        int age;
        boolean sex;
}

class Student {
        String name;
        int age;
        boolean sex;
        int score;
        String school_name;
}

class teacher {
        String name;
        int age;
        boolean sex;
        String rank;
        double salary;
        String school_name;
}

class worker{
        String name;
        int age;
        boolean sex;
        String technology;
        double salary;
        String factory;
}
```

图 5-13　四个类的创建

如图 5-13 所示，程序中声明了四个类，其中类 student、teacher 和 worker 都是人的职业，因此他们具有 person 类的成员属性 name、age 和 sex，换句话说如果新建的职业类都是人类的职业，这三个属性都需要重复地定义。

5.4.2 extends 关键字

为了解决类之间的属性重复定义的问题，面向对象的语言提供了派生机制，它允许程序员用以前已定义的类来定义一个新类。新类称作子类，原来的类称作父类或超类。两类中公共的内容放到父类中。Java 中亦有同样的机制。在 Java 中，用关键字 extends 表示派生。

语法格式如下：

```
Class 子类名 extend 父类名{
类内容
……
}
```

【例 5-10】extends 关键字使用示例如图 5-14 所示。

```
class Person{
      String name;
      int age;
      boolean sex;
}

class Student extends Person{
      int score;
      String school_name;
}

class teacher extends Person{
      String rank;
      double salary;
      String school_name;
}

class worker extends Person{
      String technology;
      double salary;
      String factory;
}
```

图 5-14　extends 关键字使用示例

如图 5-14 所示，在这段代码中，teacher 类中有 person 类的所有成员。所有这些成员都继承于父类中的定义。程序员要做的只是定义额外的特性，或者进行必要的修改。派生机制改善了程序的可维护性，增加了可靠性，提高了程序代码的复用率。对父类 person 所做的修改延伸至子类 teacher、student 和 worker 类中，而程序员不需做额外的工作。

5.4.3　单重继承

继承具有两种形式：多重继承和单继承。多继承指一个子类可以有多个父类，而单继承指一个子类只有一个父类。Java 中只支持单继承，即只允许从一个类中扩展类，因此 extends 语句后只能有一个类名。这条限制叫单重继承。Java 规定单重继承的限制，是因为它要让代码的可靠性更高。为了保留多重继承的功能，Java 提出了接口的概念。

虽然一个子类可以从父类继承所有的方法和成员变量，但它不能继承构造方法，子类想要使用父类的构造方法必须使用关键字 super。

5.5　多　　态

5.5.1　多态的概念

【例 5-11】多态程序如图 5-15 所示。

```
class Person{
        String name;
        int age;
        boolean sex;

        void run(){}
        void eat(){}
        void sleep(){}
}

class Student extends Person{
        int score;
        String school_name;
        void study(){ }
}

class teacher extends Person{
        String rank;
        double salary;
        String school_name;
        void teach(){}
}

class worker extends Person{
        String technology;
        double salary;
        String factory;
        void work(){}
}
```

图 5-15　多态程序

如图 5-15 所示，由于 teacher 类与 person 类之间具有“is a”关系，或者说，一名 teacher 也是一个 person。teacher 得到了父类 person 的所有属性，包括数据成员和方法成员。这意味着对 person 对象合法的操作，对 teacher 对象也合法。同样的道理对 worker 和 student 也一样适用。

假定 person 类中有方法 eat( )，run( )和 sleep( )，则 teacher 类亦有这两个方法。同理 student 和 worker 类也继承得到上述的三个方法这引出对象是多态的，即它们有“许多形式”。一个具体对象可以有 person 的形式，也可以有 student 的形式。

在 Java 中，有一个很特殊的类，它是所有类的父类，万类之首的 Object 类，这就是 java. lang. Object 类。事实上，前面的定义是下面定义的简写形式。

```
public class Employee extends Object
public class Manager extends Employee
```

Object 类定义了几个有用的方法，包括 toString( )。正因为有这个方法，Java 中的所有对象内容都可转换为字符串。不过，对有些对象而言，转换成字符串没有什么意义，因此很少使用 toString( )方法。

5.5.2 方法重载

如果需要在同一类中写多个方法，让它们对不同的变量进行同样的操作，就需要重载方法名。下面以一个输出文本表示的简单方法为例来说明这个问题。该方法名为 print( )。

现在假定需要打印 int、float 和 String 类型的值。每种类型的打印方式不同，这是合情合理的，因为不同的数据类型需要不同的格式，可能要进行不同的处理。此时可以建立三个方法，分别叫作 printint( )，printfloat( )和 printString( )。

在 Java 和其他几种面向对象的程序设计语言中，允许对多个方法使用同一个方法名，这就是方法名的重载。当然，前提条件是能够区分实际调用的是哪个方法，才可用这种方式。实际上，Java 根据参数来调用适当的方法。在这个例子中，根据参数自变量的类型来区分这些方法。

要重载方法名，可以有如下三种说明方法。

```
public void print(int i)
public void print(float f)
public void print(String s)
```

当调用 print 方法时，可根据自变量的类型选中相应的一个方法。

重载方法有如下两条规则。

(1) 调用语句的自变量列表必须足够判明要调用的是哪个方法。自变量的类型可能要进行正常的扩展提升(如浮点变为双精度)，但在有些情况下这会引起混淆。

(2) 方法的返回类型可能不同。如果两个同名方法只有返回类型不同，而自变量列表完全相同则是不够的，因为在方法执行前不知道能得到什么类型的返回值，因此也就不能确定要调用的是哪个方法。重载方法的参数表必须不同，即参数个数或参数类型不同。

5.5.3　覆盖

使用类的继承关系，可以从已有的类产生一个新类，在原有特性基础上，增加了新的特性，因此需要修改父类中已有的方法。如果子类中定义方法所用的名字、返回类型、参数表以及异常抛出必须和父类中方法使用的完全一样，称子类方法覆盖了父类中的方法，即子类中的成员方法将隐藏父类中的同名方法。利用方法隐藏，可以重定义父类中的方法。要注意的是，覆盖的同名方法中，子类方法不能比父类方法的访问权限更严格。例如，如果父类中方法 method( ) 的访问权限是 public，子类中就不能含有 private 的 method ( )，否则，会出现编译错误。

在子类中，若要使用父类中被隐藏的方法，可以使用 super 关键字。

【例 5-12】覆盖程序示例如图 5-16 所示。

```
class Person{
      void print(){
        System.out.println("this is a person!");
      }
}
class Student extends Person{
      void print(){
        System.out.println("this is a child!");
      }
}
class teacher extends Person{
      void print(){
        super.print();
        System.out.println("this is a teacher!");
      }
}
class worker extends Person{
      void print(){
        super.print();
        System.out.println("this is a worker!");
      }
}
public class Test{
      public static void main(String[] args){
        Person p=new Person();
        Student s=new Student();
        teacher t=new teacher();
        worker w=new worker();
        p.print();
        s.print();
        t.print();
        w.print();
      }
}
```

图 5-16　覆盖程序示例

运行结果如图 5-17 所示。

```
this is a person!
this is a child!
this is a person!
this is a teacher!
this is a person!
this is a worker!

Process completed.
```

图 5-17　覆盖程序运行结果

如图 5-16 所示，程序中创建了四个对象，每个对象都有方法 print。由于子类方法的声明和父类的完全一致，因此子类的方法覆盖了父类的方法。如果子类需要调用父类的被覆盖的方法，使用关键字 super。如果方法名相同，而参数表不同，则是对方法的重载，如图 5-17 所示。调用重载方法时，编译器将根据参数的个数和类型，选择对应的方法执行。重载和覆盖的区别在于：重载的方法属于同一个类，覆盖的方法分属于父类、子类中。

如果这样创建实例：

```
Person s=new Student( );
Person w=new worker( );
```

则 s. print( ) 和 w. print( ) 调用哪个方法就不容易弄清。执行规则是与对象真正类型(运行时类型)相关的方法，而不是与引用类型(编译时类型)相关的方法。这也是多态的另一个重要性质，常称作虚方法调用。因此，s. print( ) 将执行 Student 类中的方法，因为实例的真正类型是 Student。同理 w. print( ) 执行 worker 类中的方法。

与成员方法的覆盖一样，继承也会带来成员属性的覆盖，当子类和父类中出现同样名称的成员属性变量，子类的属性会覆盖父类的属性。

5.5.4　构造方法的重载与继承

与类其他成员方法重载一样，类构造方法也可以重载。构造方法重载是指在同一个类中定义多个不同参数的构造方法，满足创建含有不同参数的对象的使用。构造方法或形参类型不同，或形参个数不同。

例如：对于 Student 学生类，类中有两数据成员 name，score。使用不同的构造方法创建不同学生对象。

```
Student(String s,double n);
Student(String s);
Student( );
```

当类中存在多个参数不同的构造方法，创建对象时系统会自动根据对象实参的数量、数据类型和次序来选择调用构造方法。

例如：

```
Student stu1=new Student("李自成",90);          //调用第一个构造方法
```

```
Student stu2 = new Student("李白");          //调用第二个构造方法
Student stu3 = new Student( );               //调用第三个构造方法
```

同类中多个重载的构造方法也可以使用 this( ) 关键字相互调用。this( ) 关键字根据需要可以带参数，也可以不带参数。该调用语句应该出现在构造方法的第一个可执行语句的位置上。看如下实例。

【例 5-13】构造方法程序示例如图 5-18 所示。

```
class Student
{
    String name;
    double score;
    public Student(String s, double n ){
        name=s; score=n;
    }
    public Student(String s){
        this(s, 90);
    }
    public Student(double n){
        this("zhangli");
    }
}
```

图 5-18　构造方法程序

如图 5-18 所示，this( ) 关键字调用同类的其他构造方法，提高代码复用率，增强程序的抽象程度。

在构造方法的继承上，子类可以无条件地继承父类的无参构造方法。如果子类需要用到无参构造方法，父类一定要有无参的构造方法给子类继承。

【程序 5-14】构造方法的继承示例如图 5-19 所示。

```
import java.io.*;
class father
{
      public father()
             {System.out.println("it is father!");}
      public father( int x)
             {System.out.println("it is father!");}
}
class son extends father
{     public son( )
             {System.out.println("it is son!");}
}
public class Test
{ public static void main(String args[])
{
    son m=new son( ); }
}
```

图 5-19　构造方法的继承

如图 5-19 所示，如果程序中父类 father 中没有无参数构造方法 father( )，则编译器会提示出错。当然，如果子类中不需要使用无参构造方法，则父类也可以没有无参的构造方法。同时，如果子类自己没有定义构造方法，则子类将继承父类无参的构造方法作为自己的构造方法；如果子类自己定义了构造方法，则在子类创建新对象时，则先执行来自继承父类的无参构造方法，再执行自己的构造方法。

5.6 封　　装

除了保护对象的数据不受非法修改外，强制使用者通过方法来访问数据是确保方法调用后各数据仍合法的更简单的方法。例如在 Time 类的这个例子中，要计算当前时间的一个小时后，就必须通过方法得到。

如果对数据的访问是完全放开的，类的每个使用者都可以修改 second 值，并需时刻注意因 second 值的变化而引发的 hour，minute 值之间的不一致，那么，程序会变得混乱且不易控制。如果说对于大家非常熟悉的日期这种类型尚可勉强接受的话，那么对于其他复杂的数据类型，类的使用者可能会疏忽对数据的一致性检查，因此，还是让类的使用者通过方法来访问数据为好。这就是封装。封装是面向对象方法的一个重要原则。它有两个基本含义：一是指对象的全部属性数据和对数据的全部操作结合在一起，形成一个统一体，也就是对象；另一方面是指，尽可能地隐藏对象的内部细节，只保留有限的对外接口，对数据的操作都通过这些接口实现。

Java 通过四种控制符控制类型成员和方法。但是仅有三个关键词限定访问权限，包括 public，private 和 protected，如果不写访问权限，在 Java 中被称为默认权限，或同包权限，本书中以 friendly 代替。它们可以修饰类中的成分，决定所修饰成分在程序运行时被处理的方式。访问权限关键字与访问能力之间的关系见表 5-1 所列。

表 5-1　访问权限修饰符

类型	public	friendly	protected	private
同一类	yes	yes	yes	yes
同一包中的子类	yes	yes	yes	no
同一包中的非子类	yes	yes	yes	no
不同包中的子类	yes	no	yes	no
不同包中的非子类	yes	no	no	no

5.6.1 public

public 是公共访问控制符，用 public 修饰的成分表示是公有的，也就是它可以被其他任何对象访问，是权限最大的一个修饰符。public 可以修饰类、数据成员、构造方法、方法成员。Java API 提供的类都是 public 修饰的，程序可直接访问这些包。

5.6.2　friendly

friendly 是默认访问控制，也称为包权限，这个限定符不是必须写的。如果不写，则表明是“friendly”，相应的成分可以被所在包中的各类访问。类、数据成员、构造方法、方法成员都能够使用默认权限，即不写任何关键字。

5.6.3　protected

protected 保护访问控制符，即用该关键字修饰的成分是受保护的。它可以修饰数据成员、构造方法、方法成员，不能修饰类(此处指外部类，不考虑内部类)。被 protected 修饰的成员，能在定义它们同包的类中被调用。如果有不同包的类想调用它们，那么这个类必须是定义它们的类的子类。

5.6.4　private

private 可以修饰数据成员、构造方法、方法成员，不能修饰类(此处指外部类，不考虑内部类)。private 是私有访问控制符，和它的名字“私有”一样，类中限定为 private 的成员只能被这个类本身访问，在类外不可见。

5.6.5　继承与封装

类的访问控制机制丰富了类的封装特性，程序员根据程序设计的要求限制使用者对类成员的访问机制，保护数据不会被任意地修改和使用。通过访问控制机制也使得程序员必须设计合理有效的接口来操纵成员方法，使得数据处理流程被隐藏起来。

类的继承机制使得程序员很容易根据现有类扩展新类，而且程序员并不一定需要详细了解父类的内部构造情况。

因此 Java 的继承与封装机制有效地保护了类成员的信息隐蔽性问题，防止对类成员的任意修改，同时也能够保证类的可扩展性。

5.7　抽象类、内部类、接口

Java 也提供了一些独特的类形式设计，包括抽象类、内部类和接口。

5.7.1　抽象类

Java 程序用抽象类(abstract class)来实现自然界的抽象概念。在程序设计过程中，有时需要创建某个类代表一些基本行为，并为其定义一些方法，但是又无法或不宜在这个类中就对这些行为加以实现，而希望在其子类中再去实现这些方法。例如，设计一个名为 shape 的类，它代表了不同图形，每种图形都有周长和面积，然而根据图形的不同，周长和面积的计算也不相同。因而只在 shape 类中定义周长和面积，其计算的详细过程放在子类中执行。

像 shape 类这种定义了方法但没有定义具体实现的类通常称为抽象类(abstract class)。

在 Java 中可以通过关键字 abstract 把一个类定义为抽象类，每一个未被定义具体实现的方法也应标记为 abstract(可以称为抽象方法)。

【例 5-14】抽象类的示例如图 5-20 所示。

```
abstract class Shape{
      double area;
      double girth;
      abstract public double  getArea();
      abstract public double getGirth();
}
```

图 5-20　抽象类的示例

【例 5-15】抽象类的实现如图 5-21 所示。

```
abstract class Shape{
      double area;
      double girth;
      abstract public double  getArea();
      abstract public double getGirth();
}
class circle extends Shape{
      int r;
      double  pi=3.1415;
      public double  getArea(){
        return area=r*r*pi;
      }
      public double getGirth(){
        return girth=2*r*pi;
      }
}
class square extends Shape{
     int length;
     int width;
      public double  getArea(){
        return area=length*width;
      }
      public double getGirth(){
        return girth=2*(length+width);
      }

}
```

图 5-21　抽象类的实现

如图 5-20 所示，类 shape 中只是给出了抽象方法 getArea( ) 和 getGirth( )，并没有实现，因此使用 abstract 关键词修饰。而子类 cirlce 和 squre 继承类 shape，它们不是抽象方法，所以必须要实现父类的抽象方法，如图 5-21 所示。

抽象类的作用在于将许多有关的类组织在一起，提供一个公共的类，即抽象类，而那些被它组织在一起的具体的类将作为它的子类由它派生出来。抽象类刻画了公有行为的特征，并通过继承机制传送给它的派生类。在抽象类中定义的方法称为抽象方法，这些方法只

有方法头的声明，而用一个分号来代替方法体的定义，即只定义成员方法的接口形式，而没有具体操作。只有派生类对抽象成员方法的重定义才真正实现与该派生类相关的操作。

在各子类继承了父类的抽象方法之后，再分别用不同的语句和方法体来重新定义它，形成若干个名字相同、返回值相同、参数列表相同、目的一致但是具体实现有一定差别的方法。抽象类中定义抽象方法的目的是实现一个接口，即所有的子类对外都呈现一个相同名字的方法。抽象类是它的所有子类的公共属性的集合，是包含一个或多个抽象方法的类。使用抽象类的一大优点就是可以充分利用这些公共属性来提高开发和维护程序的效率。对于抽象类与抽象方法的限制总结如下。

（1）凡是用 abstract 修饰符修饰的类被称为抽象类，凡是用 abstract 修饰符修饰的成员方法被称为抽象方法。

（2）抽象类中可以有零个或多个抽象方法，也可以包含非抽象的方法。

（3）抽象类中可以没有抽象方法，但是，有抽象方法的类必须是抽象类。

（4）对于抽象方法来说，在抽象类中只指定其方法名及其类型，而不书写其实现代码。

（5）抽象类可以派生子类，在抽象类派生的子类中必须实现抽象类中定义的所有抽象方法。

（6）抽象类不能创建对象，创建对象的工作由抽象类派生的子类来实现

（7）如果父类中已有同名的 abstract 方法，则子类中就不能再有同名的抽象方法。

（8）abstract 不能与 final 并列修饰同一个类。

（9）abstract 不能与 private，static，final 或 native 并列修饰同一个方法。

5.7.2　内部类

内部类也称嵌套类。JDK 支持以一个类作为其他类的成员，既可以在语句块中局部定义也可以在表达式中匿名定义。内部类的建立如下：

```
class Outer {
    class Inner{ }
    }
```

编译上述代码会产生两个文件：Outer. class 和 Outer $ Inner. class。

内部类具有以下几个属性。

（1）类名只能在定义的范围内被使用，内部类的名称必须区别于外部类。

（2）内部类是外部类的一个成员，内部类可以使用外部类的类变量和实例变量，也可以使用外部的局部变量，无论是否是 private 的，内部类不能用普通的方式访问。

（3）内部类可以定义为 abstract 类型。

（4）内部类也可以是一个接口（这时已很难说它是类还是接口了），这个接口必须由另一个内部类来实现。

（5）内部类可以被定义为 private 或 protected 类型。当一个类中嵌套另一个类时，访问保护并不妨碍内部类使用外部类的成员。

（6）被定义为 static 型的内部类将自动转化为顶层类，它们不能再使用局部范围中或其他内部类中的数据和变量。

（7）内部类不能定义 static 型成员，而只有顶层类才能定义 static 型成员。如果内部类

需要使用 static 型成员，这个成员必须在外部类中加以定义，否则编译器会提示错误。

```
class Outer {
class Inner{
        static int a=10;
    }}
```

（8）内部类仍然是一个独立的类，在编译之后内部类会被编译成独的 .class 文件，但是前面冠以外部类的类命和 $ 符号。

（9）从外部类的非静态方法中实例化内部类对象，参看【例 5-16】，如图 5-22 所示。

【例 5-16】

```
class Outclass {
    private int i = 10;
    public void makeInnerClass(){
        Innerclass in = new Innerclass();
        in.print();
    }
    class Innerclass{
        public void print(){
            System.out.print(i);
        }
    }
}
```

图 5-22　内部类程序

也可以通过从外部类的静态方法中实例化内部类对象，参看【例 5-17】，如图 5-23 所示。

【例 5-17】

```
class Outclass {
    private int i = 10;
    public void makeInnerClass(){
       Innerclass in = new Innerclass();
       in.print();
    }
    class Innerclass{
       public void print(){
       System.out.print(i);
       }
    }
}
public class Test
{ public static void main(String args[])
  {
    Outclass out=new Outclass();
    Outclass.Innerclass inner=out.new Innerclass();
    inner.print();
}}
```

图 5-23　外部类的实例化内部类对象程序

5.7.3　匿名类

匿名类是不能有名称的类，所以没办法引用它们。必须在创建时，作为 new 语句的一部分来声明它们。其声明格式如下：

new<类或接口><类的主体>

这种形式的 new 语句声明一个新的匿名类，它对一个给定的类进行扩展，或者实现一个给定的接口。它还创建那个类的一个新实例，并把它作为语句的结果而返回。要扩展的类和要实现的接口是 new 语句的操作数，后跟匿名类的主体。假如匿名类对另一个类进行扩展，它的主体可以访问类的成员，覆盖它的方法等，这和其他任何标准的类都是一样的。假如匿名类实现了一个接口，它的主体必须实现接口的方法。注重匿名类的声明是在编译时进行的，实例化在运行时进行。这意味着 for 循环中的一个 new 语句会创建相同匿名类的几个实例，而不是创建几个不同匿名类的一个实例。从技术上说，匿名类可被视为非静态的内部类，所以它们具有和方法内部声明的非静态内部类一样的权限和限制。假如要执行的任务需要一个对象，但却不值得创建全新的对象(原因可能是所需的类过于简单，或者是由于它只在一个方法内部使用)，匿名类就显得非常有用。匿名类尤其适合在 Swing 应用程序中快速创建事件处理程序。

匿名类重写了类的方法，参看【例 5-18】，如图 5-24 所示。

【例 5-18】

```
public class AnonymousClass {
    public static void main(String[] args) {
        Teacher zhang = new Teacher();
        zhang.look(new Student(){
            void speak(){
                System.out.println("I");
            }
        });
    }

}
 class Teacher{
    void look(Student stu){
        stu.speak();
    }
}
 class Student{
    void speak(){
        System.out.println("I am a student!");
    }
}
```

图 5-24　匿名类程序

匿名类实现了接口的方法，参看【例 5-19】，如图 5-25 所示。

【例 5-19】

```
public class AnonymousClass {
    public static void main(String[] args) {
        A a = new A();
        a.f(new Show(){
            public void show(){
                System.out.println("实现了接口的匿名类");
            }
        });
    }
}

class A{
    void f(Show s){
        s.show();
    }
}
interface Show{
    public void show();
}
```

图 5-25　匿名类实现了接口程序

5.7.4　接口

接口是抽象类功能的另一种实现方法。在接口中所有的方法都是抽象方法，都没有方法体。从这个角度上讲，也可以把接口看成特殊的抽象类。然而，接口还可以实现与抽象类不同的功能。Java 不支持多重继承的概念，一个类只能由唯一的一个类继承而来。这并不意味着 Java 不能实现 C++中多重继承的功能。事实上，在 Java 中定义了接口的概念，Java 允许一个类同时实现(implements)多个接口，从而具有和多重继承同样强大的能力，并具有更加清晰的结构。

接口的定义形式为

[修饰符]　interface　接口名{

　　方法原型或静态常量

}

在【例 5-20】中，仿照上一小节的程序，使用接口的方式重新定义了一个存储字符的数据结构。

【例 5-20】接口程序示例如图 5-26 所示。

```
interface Shape{
      abstract public double  getArea();
      abstract public double getGirth();
}
class circle implements  Shape{
      double area;
      double girth;
      int r;
      double  pi=3.1415;
      public double  getArea(){
        return area=r*r*pi;
      }
      public double getGirth(){
        return girth=2*r*pi;
      }
}
class square implements Shape{
     double area;
     double girth;
     int length;
     int width;
      public double  getArea(){
        return area=length*width;
      }
      public double getGirth(){
        return girth=2*(length+width);
      }
}
```

图 5-26　接口程序

如图 5-26 所示，Shape 是一个接口，具有两个抽象方法，类 square 和 circle 实现了接口 Shape 中定义的抽象方法 getArea( ) 和 getGirth( )。要注意的是，在接口中定义的成员变量都默认为终极类变量，即系统会将其自动增加 final 和 static 这两个关键字，并且对该变量必须设置初值。考虑到这点，程序中类 square 和 circle 各自声明自己的 area 和 girth 成员变量。

接口的实现与类的继承是相似的，不同之处是：实现接口的类不从该接口的定义中继承任何行为，在实现该接口的类的任何对象中都能够调用这个接口中定义的方法。在实现的过程中，这个类还可以同时实现其他接口。要实现接口，可在一个类的声明中用关键字 implement，表示该类已经实现的接口。

在 Java 中，可以通过在 implements 后面声明多个接口名来同时实现多个接口，如图 5-27 所示。

【例 5-21】多接口实现程序示例如图 5-27 所示。

```
interface interface1{
    abstract public int method1();
    abstract public int method2();
}

interface interface2{
    abstract public int method3();
    abstract public int method4();
}

interface interface3{
    abstract public int method5();
    abstract public int method6();
}

class class1 implements interface1,interface2,interface3{
    public int method1(){return 0;};
    public int method2(){return 0;};
    public int method3(){return 0;};
    public int method4(){return 0;};
    public int method5(){return 0;};
    public int method6(){return 0;};
}
```

图 5-27　多接口实现程序

另外，同抽象类一样，使用接口名称作为一个引用变量的类型也是允许的，该引用可以用来指向任何实现了该接口的类的实例。使用时将根据动态连接的原则，视该变量所指向的具体实例进行操作。如：

```
Shape   t=new circle();
Shape   s=new square();
```

5.8　关键字 this

在传统的函数中，可以使用函数的命名变量来表示要对之施加操作的数据。在 Java 中，如果在类的成员方法中访问类的成员变量，可以使用关键字 this 指明要操作的对象，如图 5-28 所示。

【例 5-22】this 程序如图 5-28 所示。

```
public class Person{
    String name;
    int age;
    int weight;
    int height;
    int ID;
    boolean sex;

    void Print(){
        System.out.println(this.age);
        System.out.println(this.name);
        System.out.println(this.ID);
        System.out.println(this.height);
        System.out.println(this.weight);
    }
}
```

图 5-28　this 程序

在类方法中，如图 5-28 所示，Java 自动用 this 关键字把所有变量和方法引用结合在一起。该例中 this 的使用是不必要的，程序中可以不写该关键字。有些情况下关键字 this 是必需的。例如，在完全独立的类中调用一个方法，同时把对象实例作为一个自变量来传送，此时要用 this 指明对哪个对象实例进行操作。

【例 5-23】this 作为参量的程序如图 5-29 所示。

```
class My {
      public My() {
        new Your(this).print();
      }

      public void print() {
        System.out.println("Hello from My!");
      }
}

class Your {
    My a;
    public Your(My a) {
       this.a = a;
    }

    public void print() {
       a.print();
       System.out.println("Hello from Your!");
    }

    public static void main(String[] args){
        My a=new My();
        Your b=new Your(a);
    }
}
```

图 5-29　this 作为参量程序

运行结果如图 5-30 所示。

```
Hello from My!
Hello from Your!

Process completed.
```

图 5-30　运行结果

5.9　关键字 final

final 是 Java 中一个重要的关键字，不仅可以修饰一个类，也可修饰类中的成员变量和成员方法。凡是使用该关键字修饰的类或类成员都是不能改变的。

5.9.1　最终成员

如果使用 final 修饰一个变量，则该变量称为常量，也称为最终成员。

【例 5-24】最终变量程序如图 5-31 所示。

```
import java.io.*;
class circle{
      public final double pi=3.1415;
      public int r=0;
      public double girth=2*pi*r;

}

public class Test
{ public static void main(String args[])
{
     circle c=new circle();
     c.pi=3.1415926;
}}
```

图 5-31　最终变量程序

运行结果如图 5-32 所示。

```
.0_05 <Default> - <Default>--------------------
D:\Backup\我的文档\JCreator Pro\MyProjects\Test\Test.java
:13: 错误: 无法为最终变量pi分配值
     c.pi=3.1415926;
      ^
1 个错误

Process completed.
```

图 5-32　最终变量程序运行结果

如图 5-31 和图 5-32 所示，通过 finial 将变量改为常量，不仅可以丰富系统定义的常数，可以通过精度等方面进行必要的调整，而且保证常量使用的统一，并为修改提供方便。简单类型的变量经过 finial 修饰后变为常量，而类类型的变量，或者说将一个引用类型的变量标记为 final，那么这个变量将不能再指向其他对象，但它所指对象的取值仍然是可以改变的。

【例 5-25】最终对象程序如图 5-33 所示。

```
import java.io.*;
class circle{
      public final double pi=3.1415;
      public int r=0;
      public double girth=2*pi*r;

}

public class Test
{ public static void main(String args[])
{
     final circle c=new circle();
     c.r=10;
     c=new circle();
}}
```

图 5-33　最终对象程序

运行结果如图 5-34 所示。

```
.0_05 <Default> - <Default>--------------------
D:\Backup\我的文档\JCreator Pro\MyProjects\Test\Test.java
:14: 错误: 无法为最终变量c分配值
     c=new circle();
     ^
1 个错误

Process completed.
```

图 5-34　最终对象程序运行结果

如图 5-33 所示，程序中，类 circle 具有一个对象 c，c 的成员变量指向 r 的值是可以的，但如果试图用 c 指向其他对象就会引起错误，如图 5-34 所示。

5.9.2　最终方法

被标记为 final 的成员方法称为最终方法(final method)，被标记 final 的方法将不能被覆盖。

终极方法的定义格式为

```
final returnType finalMethod([parameter list]){
     ……
}
```

如图 5-35 所示，在例子的程序片段中，由于试图覆盖终极方法，因此将会出现错误，如图 5-36 所示。

【例 5-26】最终方法程序如图 5-35 所示。

```
class father{
    final void print(){System.out.println("it is a fatehr!");}
}

class sun extends father{
    final void print(){System.out.println("it is a sun!");}
}
```

图 5-35　最终方法程序

运行结果如图 5-36 所示。

```
.0_05 <Default> - <Default>--------------------
D:\Backup\我的文档\JCreator Pro\MyProjects\Test\Test.java
:15: 错误: sun中的print()无法覆盖father中的print()
    final void print(){System.out.println("it is a sun!")
;}
                  ^
  被覆盖的方法为final
1 个错误
```

图 5-36　最终方法程序运行结果

5.9.3　最终类

出于设计的需要，有时候一些类是不能被继承的。例如，Java. lang. String 就是如此。这样做的目的是为了保证如果一个方法中有一个指向 String 类的引用，那么它肯定就是一个真正的 String 类型，而不是一个已被更改的 String 的子类。另外一种情况是某个类的结构和功能已经很完整，不需要生成它的子类，这时也应该在这个类的声明中以关键字 final 进行修饰。被标记为 final 的类将不能被继承，这样的类可以称之为终极类，其声明的格式为

```
final class finalClassName{
    ……
}
```

如图 5-37 所示，【例 5-27】中定义了一个 final 类，然后试图派生它的子类，结果将导致错误，如图 5-38 所示。

【例 5-27】最终类程序如图 5-37 所示。

```
final class father{
    void print(){System.out.println("it is a fatehr!");}
}

class sun extends father{
    void print(){System.out.println("it is a sun!");}
}
```

图 5-37　最终类程序

运行结果如图 5-38 所示。

```
.0_05 <Default> - <Default>--------------------
D:\Backup\我的文档\JCreator Pro\MyProjects\Test\Test.java
:14: 错误: 无法从最终father进行继承
class sun extends father{
                  ^
1 个错误

Process completed.
```

图 5-38　最终类程序运行结果

5.10　关 键 字 static

static 既可修饰数据成员，又可以修饰成员方法，表明所说明的对象是静态的，提供对象共享相同类成员属性机制。

5.10.1　静态成员

静态成员与类相对应，它可以被类的所有对象共享，定义了类之后即已存在。类中定义的公有静态变量相当于全局变量。

【例 5-28】静态成员属性程序如图 5-39 所示。

```
class count{
      public int x=0;
      static public int y=0;
      count(){
        x++;
        y++;
      }

      public void print(){
        System.out.println(x);
        System.out.println(y);
      }
}
public class Test
{ public static void main(String args[])
 {
      count a=new count();
      a.print();
      count b=new count();
      b.print();
      count c=new count();
      c.print();

}}
```

图 5-39　静态成员属性程序

运行结果如图 5-40 所示。

```
.0_05 <Default> - <Default>--
1
1
1
2
1
3

Process completed.
```

图 5-40　静态成员属性程序运行结果

如图 5-39 所示，类 Count 中定义了私有成员 x 和 y，但 y 是静态变量，三个对象共享该静态成员，因此，每个对象创建的时候，都会对 y 值累加，而 x 变量是每个成员独有的，因此每次对象创建时，都会有一个独立的 x 值创建，值都为 1，如图 5-40 所示。不仅可以通过对象调用静态变量，也可以通过类名调用静态变量，结果是一致的，如 c. y = 10 或 count. y = 20。

5. 10. 2　静态方法

静态方法与一般方法在使用上是有所区别的，静态方法可以不需要建立类的实例就可以直接调用，但是一般方法需要建立实例。静态的方法和变量会调用时在内存生成一个唯一的标示，调用时候可以直接找到，而且会节省内存，但是如果静态方法过多的话，有可能会报内存溢出。

【例 5-29】静态方法程序如图 5-41 所示。

```
public class Test
{
    static void  print(){
       System.out.println("this is a static method.");
    }
    public static void main(String args[])
    {
       Test.print();

    }
}
```

图 5-41　静态方法程序

5. 11　关 键 字 super

super 关键字与 this 关键字都可以实现将屏蔽的类成员为可见。如果父类的方法被子类的方法重写，则使用如下的语法结构调用父类被隐藏的方法。

super. <父类成员变量/方法>

如果子类想调用父类的构造方法，则使用如下的语法结构：

super(参数)

同时需要注意，子类调用父类的构造方法时，super 语句必须是子类构造方法的第一句。

【例 5-30】super 调用父类构造方法，如图 5-42 所示。

```
public class superclass {
    int a;
    int b;
    int c;
    public superclass(int x,int y,int z) {
        a=x;
        b=y;
        c=z;
    }
}
class subclass extends superclass{
    int d;
    int e;
    int f;
    subclass(int x,int y,int z,int h,int i,int j){
        super(x,y,z);
        d=h;
        e=i;
        f=j;
    }
}
```

图 5-42　super 调用父类构造方法

本 章 小 结

类是客观世界的抽象，对象代表现实世界的实体，如学生、交通工具、银行账号等，对象通过类描述实体的状态和行为。通过类将数据和方法封装，建立对象的内部机制。继承、多态和封装使得程序语言设计更加方便和安全。针对继承机制而带来的冲突，通过覆盖、重载和限定符进行解决。一些关键字如 this，super，final 和 static 等丰富了 Java 的语法结构，使得程序设计更为灵活。

思考及练习题

1. 面向对象语言的基本特征是什么？
2. 试述什么是类，什么是对象，类和对象的关系。
3. 什么是方法？方法的作用、方法的递归调用如何实现？
4. 什么是单重继承？什么是多重继承？Java 采用什么继承？
5. Java 语言中，被 static 修饰的成员变量和类变量有什么区别？
6. 关键字 protected 的作用是什么？在什么情况下适用？
7. 什么是接口，为什么要定义接口？接口与类有何异同，使用什么关键字实现？
8. 编写程序，定义一个名为 Rectangle 表示长方形的类，它包含 double 类型的长 length 和宽 width 两个数据成员，和设置长方形长宽值的方法 setdim( )、计算长方形面积的方法 area( )，分别计算长方形的面积。
9. 编写程序，设计一个 person 类，其中包含 teacher、student 子类，完成教师和学生信息的插入、删除、查找功能。