第 5 章 类的高级特性

CHAPTER

5.1 static 关键字

static 关键字用来声明静态变量和静态方法。例如：

```
class MyClass {
    static int i;
    static void increase(){
        i++;
    }
}
```

静态变量和静态方法为类中所有对象所公有，可以不创建对象，直接引用，也称为类变量和类方法。

引用方式：类名.静态变量/静态方法，如：

```
MyClass.i;
MyClass.increase();
```

如果在声明时不用 static 关键字修饰，则为实例变量和实例方法。

一个类通过使用 new 运算符可以创建多个不同的对象，这些对象将被分配不同的内存空间，准确地说，就是不同的对象的实例变量将被分配不同的内存空间，如果类中的成员变量有类变量，所有的对象的这个类变量都分配给相同的一处内存。也就是说，对象共享类变量，改变其中一个对象的这个类变量会影响其他对象的这个类变量。

静态变量可以通过类名直接访问，也可以通过对象来调用。采用这两种方法取得的结果是相同的。如果是 public 静态变量，则其他类可以不通过实例化访问它们。

类方法不能访问实例变量，只能访问类变量。类方法可以由类名直接调用，也可由实例对象进行调用。类方法中不能使用 this 或 super 关键字。

对于实例变量必须先生成实例对象，通过该对象访问实例变量。实例方法可以对当前对象的实例变量进行操作，也可以对类变量进行操作，实例方法由实例对象调用。下面的代码及图 5-1 说明了实例变量与静态变量的关系。

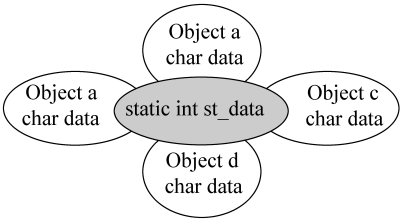

图 5-1　实例变量与静态变量关系

```
class ABCD {
    char data;
    static int st_data;
}
class Demo {
    ABCD a,b,c,d
}
```

例 5-1　关于实例成员和类成员的例子。

程序清单：ch05\MemberTest.java

```
package ch05;
    class Member {
      static int classVar;
      int instanceVar;
      static void setClassVar(int i) {
        classVar=i;
        //instanceVar=i;          //类方法不能访问实例变量
      }
      static int getClassVar(){
          return classVar;
      }
      void setInstanceVar(int i){
          classVar=i;             //实例方法不但可以访问类变量,也可以访问实例变量
          instanceVar=i;
      }
      int getInstanceVar() {
          return instanceVar;
    }
    }
public class MemberTest{
  public static void main(String[] args) {
    Member m1=new Member();
    Member m2=new Member();
    m1.setClassVar(1);
    m2.setClassVar(2);
    System.out.println("m1.classVar="+m1.getClassVar()+" m2.ClassVar="+m2.
    getClassVar());
    m1.setInstanceVar(11);
    m2.setInstanceVar(22);
```

```
        System.out.println("m1.InstanceVar="+m1.getInstanceVar()+" m2.InstanceVar=
        "+m2.getInstanceVar());
        }
}
```

程序运行结果如下：

```
m1.classVar=2 m2.ClassVar=2
m1.InstanceVar=11 m2.InstanceVar=22
```

分析一个不正确的变量引用实例：

```
class StaticError{
  String mystring="hello";                          //实例变量
    public static void main(String[] args) {
        System.out.println(mystring);               //静态方法访问实例变量出错
  }
}
```

错误信息：can't make a static reference to nonstatic variable。因为只有对象的方法可以访问对象的变量。

解决的办法如下。

(1) 将实例变量 mystring 改为类变量：

```
class StaticError{
    static String mystring="hello";
    public static void main(String[] args) {
        System.out.println(mystring);
    }
}
```

(2) 将实例变量 mystring 改为局部变量：

```
class NoStaticError{
    public static void main(String[] args) {
        String mystring="hello";
        System.out.println(mystring);
    }
}
```

例 5-2　下面例子中的梯形对象共享一个 static 的下底。

程序清单：ch05\CommonLader.java

```
package ch05;
    class 梯形{
        float 上底,高;                                  //类的变量
    static float 下底;                                  //类的变量
    梯形(float x,float y,float h) {                     //构造方法
```

```
        上底=x; 下底=y; 高=h;
    }
    float 获取下底() {
        return 下底;
    }
    void 修改下底(float b) {
        下底=b;
    }
}
public class CommonLader{
    public static void main(String[] args){
        梯形 laderOne=new 梯形(3.0f,10.0f,20);
        梯形 laderTwo=new 梯形(2.0f,3.0f,10);
        System.out.println("laderOne 的下底:"+laderOne.获取下底());
        System.out.println("laderTwo 的下底:"+laderTwo.获取下底());
        laderTwo.修改下底(60);
        System.out.println("laderOne 的下底:"+laderOne.获取下底());
        System.out.println("laderTwo 的下底:"+laderTwo.获取下底());
    }
}
```

程序运行结果如下：

```
laderOne 的下底: 3.0
laderTwo 的下底: 3.0
laderOne 的下底: 60.0
laderTwo 的下底: 60.0
```

当 Java 程序执行时，类的字节码文件被加载到内存，如果该类没有创建对象，类的实例成员变量不会被分配内存。但是，类中的类变量，在该类被加载到内存时，就分配了相应的内存空间。如果该类创建对象，那么不同对象的实例变量互不相同，即分配不同的内存空间，而类变量不再重新分配内存，所有的对象共享类变量，即所有的对象的类变量是相同的一处内存空间，类变量的内存空间直到程序退出运行，才释放所占有的内存。

实例方法和类方法的区别如下。

对于类中的类方法，在该类被加载到内存时，就分配了相应的入口地址。从而类方法不仅可以被类创建的任何对象调用执行，也可以直接通过类名调用。类方法的入口地址直到程序退出才被取消。

当类的字节码文件被加载到内存时，类的实例方法不会被分配入口地址。当该类创建对象后，类中的实例方法才分配入口地址，从而实例方法可以被类创建的任何对象调用执行。需要注意的是，当创建第一个对象时，类中的类方法就分配了入口地址，当再创建对象时，不再分配入口地址，也就是说，方法的入口地址被所有的对象共享，当所有的对象都不存在时，方法的入口地址才被取消。

无论是类方法还是实例方法，当被调用执行时，方法中的局部变量才被分配内存空

间，方法调用完毕，局部变量即刻释放所占的内存。在一个方法被调用执行完毕之前，如果该方法又被调用，那么，方法的局部变量会再次被分配新的内存空间，例如，方法在递归调用时，方法中的局部变量会再次被分配新的内存空间。

5.2　this 关键字

this 关键字可以出现在类的实例方法中，代表使用该方法的当前对象。

为了说明 this 的用法，下面例子中的“三角形”的构造方法中，有意使用了 this。当使用构造方法来创建对象时，构造方法中的 this 就代表当前对象。

例 5-3　“三角形”的构造方法中的 this 就代表当前对象。

程序清单：ch05\Triangle.java

```
package ch05;
  class 三角形{
      double a,b,c;
      三角形(double a,double b,double c) {                    //构造方法
          setABC(this,a,b,c);                                  //调用实例方法
      }
      void setABC(三角形 trangle,double a,double b,double c) {
                                                  //实例方法,供构造方法调用初始化实例对象
          trangle.a=a;
          trangle.b=b;
          trangle.c=c;
      }
  }
  class Triangle{
      public static void main(String[] args) {
        三角形 tra=new 三角形(3,4,5);
        System.out.print("三角形的三边是:"+tra.a+","+tra.b+","+tra.c+",");
      }
  }
```

运行结果如下：

```
三角形的三边是: 3.0,4.0,5.0,
```

实例方法可以操作类的成员变量，实际上，当成员变量在实例方法中出现时，默认的格式是“this.成员变量”。如：

```
class A{
    int x,y;
    void fun(int x){
        this.x=x;
        y=x;
```

```
    }
}
```

在上述 A 类中的实例方法 fun()中出现了 this，this 就代表使用 fun()的当前对象。所以，“this.x”就表示当前对象的变量 x，当对象调用方法 fun()时，将函数的局部变量 x 赋给该对象的变量 x。当一个对象调用方法时，方法中的成员变量就是指分配给该对象的成员变量。

通常情况下，可以省略成员变量名字前面的“this.”，如成员变量 y。但是，当成员变量的名字和局部变量的名字相同时，成员变量前面的“this.”就不可以省略。

同样，类的实例方法可以调用类的其他方法，调用的默认格式是“this.方法”，如：

```
class B{
    void fun() {
        this.get();                                    //可省略 this.
    }
    void get() {
        System.out.println("ok");
    }
}
```

在上述 B 类中的方法 fun()中出现了 this，this 代表使用方法 fun()的当前对象。所以，方法 fun()的方法体中 this.get()就是调用当前对象的方法 get()。也就是说，当某个对象调用方法 fun()的过程中，又调用了方法 get()。由于这种逻辑关系非常明确，一个方法调用另一个方法时可以省略方法名字前面的“this.”。

但是，this 不能出现在类方法中，因为类方法可以通过类名直接调用，这时可能还没有任何对象诞生。

例 5-4 创建一个有两个方法的类，其中第一个方法使用 this，第二个方法不使用 this。

程序清单：ch05\Rectangle.java

```
package ch05;
    class Rectangle{                                   //矩形类
        int width;                                     //矩形的宽
        int usethis(int width){                        //返回宽度的函数
        this. width=width;                             //this 指自己这个对象
        return width;
        }
    int unusethis(int width){
            int w=width;
        return w;
    }
public static void main(String[] args){
    Rectangle r=new Rectangle();                       //类对象的实例化
    System.out.println("r.width="+r.width+" r.usethis(1)="+r.usethis(1)+
```

```
    "r.width="+r.width);
    System.out.println("r.width="+r.width+" r.unusethis(2)="+r.unusethis(2)
    +" r.width="+r.width);
    }
}
```

运行结果如下：

```
r.width=0 r.usethis(1)=1 r.width=1
r.width=1 r.unusethis(2)=2 r.width=1
```

例 5-5　编译并运行下面的程序，分析运行结果，进一步了解 super 和 this 的作用。

程序清单：ch05\SuperDemo.java

```
package ch05;
    mport java.io.*;
    class SuperClass{                                    //定义父类
        int x;
        SuperClass(){                                    //父类的构造方法
            x=10;
        }
        void doClass(){
            System.out.println("SuperClass.doClass()");
        }
    }
    class SubClass extends SuperClass{                   //定义子类
        int x;
        SubClass(){                                      //子类的构造方法
          super();                                       //调用父类的构造方法
          x=100;
        }
        void doClass(){                                  //重写父类的 doClass 方法
            System.out.println("SubClass.doClass()");
        }
        void doDemo(){                                   //演示 super 和 this 的方法
            int x;
            x=1000;
            super.doClass();                             //调用父类的 doClass 方法
            doClass();                                   //调用本类的 doClass 方法
            System.out.println("super.x="+super.x);      //父类的 x
            System.out.println("this.x="+this.x);        //本类的 x
            System.out.println("x="+x);                  //本方法的 x
        }
}
public class SuperDemo{
    public static void main(String[] args){              //主方法
```

```
        SubClass s=new SubClass();
        s.doDemo();
    }
}
```

运行结果如下：

```
SuperClass.doClass()
SubClass.doClass()
super.x=10
this.x=100
x=1000
```

分析：此程序中定义了一个父类，子类 SubClass 继承了父类 SuperClass，在主函数中定义 SubClass 类对象 s 时，自动调用类 SubClass 的构造函数 SubClass()，在此构造函数中先执行“super() ;”语句，这样就调用类 SuperClass 的构造方法 SuperClass()，因为 super 来指明超类中的方法。同样在子类方法 doDemo()中，执行语句“super.doClass();”时，则调用父类的方法 doClass()。如不用 super 来指定，则调用的是子类的方法 doClass()，这里子类 SubClass()的成员方法 doClass()重写了父类 SuperClass()的成员方法 doClass()。

语句“System. out. println (" super. x = " + super. x);”中 super 调用的是父类 SuperClass 的变量 x，而语句“System.out.println("this.x=" +this.x);”中 this 调用子类 SubClass 的变量 x，关键字 this 和 super 分别用来指明子类和父类中同名的成员变量。这里父类 SuperClass 的成员变量 x、子类 SubClass 的成员变量 x 和类方法 doDemo()中使用的局部变量 x 三者同名，则要使用关键字 this 和 super 来指定所要使用的变量。如不用则输出类方法的局部变量，如语句“System.out.println("x=" +x);”输出的就是类方法 doDemo()的局部变量。这里子类 SubClass()的成员变量 x 隐藏了父类 SuperClass()的成员变量 x。

5.3 静态导入

在 JDK 1.5 后新加了导入静态方法和静态域的功能。在类中使用静态导入，可以使用其他类中定义的类方法和类变量，而且这些类方法和类变量就像在本地定义的一样。换句话说，静态导入允许在调用其他类中定义的静态成员时，可以不通过类名直接使用类中的静态方法。

导入一个类一般都用 import xxx.ClassName；而静态导入语句是 import static xxx.ClassName.* ；这里多了 static 和.* ，意思是导入 xxx.ClassName 这个类里的所有静态方法和静态域。当然，也可以只导入某个静态方法，只要把 .* 换成静态方法名就行了。以后在这个类中，就可以直接用方法名调用静态方法，而不必用“ClassName.方法名”的方式来调用。

静态导入语句看起来和普通的 import 语句非常相似。但是，普通 import 语句从某

个包中导入的是一个或所有的类，而静态 import 语句从某个类中导入的却是一个或所有的类方法以及类变量。需要注意的是，针对一个给定的包，不可能用一行语句静态地导入所有类的所有类方法和类变量。也就是说，不能这样编写代码：

```
import static java.lang.*;          //编译出错!
```

如果一个本地方法和一个静态导入的方法有着相同的名字，那么本地方法将被优先调用。

例如，源文件顶部添加：import static java.lang.System.*；那么就可以使用 System 类的静态方法和静态域，而不必加类前缀。如：

```
out.println("hello world");       //相当于 System.out.println("hello world");
exit(0);                          //相当于 System.exit(0);
```

导入静态方法和导入静态域有以下两个实际的应用。

(1) 算术函数，对 Math 类使用静态导入，就能更自然地使用算术函数。如：

```
sqrt(pow(x,2)+pow(y,2));
```

(2) 笨重的常量，如果需要使用大量带有冗长名字的常量，就应该使用静态导入。如：

```
Date d=new Date();
if (d.get(DAY_OF_WEEK)==MONDAY)
```

看起来比 if (d.get(Calendar.DAY_OF_WEEK)==Calendar.MONDAY)清晰。

例 5-6　导入静态方法举例，计算直角三角形的斜边。

程序清单：ch05\StaticImportTest.java

```
package ch05;
    import static java.lang.Math.sqrt;                    //导入静态方法 sqrt()
    import static java.lang.Math.pow;                     //导入静态方法 pow()
    class Hypot {
        public static void main(String args[]) {
          double side1, side2;
          double hypot;
          side1=3.0;
          side2=4.0;
          //静态导入后,不再需要通过类名 Math 来调用方法 sqrt()和 pow()
          hypot=sqrt(pow(side1, 2)+pow(side2, 2));
          System.out.println("给定 RT△的两边边长为:" +side1+"和"+side2+", 其斜
          边边长为:"+hypot);
        }
    }
```

程序运行结果如下：

```
给定 RT△的两边边长为: 3.0 和 4.0, 其斜边边长为: 5.0
```

5.4 final 关键字

final 关键字可以修饰类、类的成员变量和成员方法，但 final 的作用不同。

(1) final 修饰成员变量，则成员变量成为常量。为了声明一个 final 变量，可以在类型之前的变量声明使用 final 关键字，例如：

```
final type piVar=3.14159;
```

可以在任何作用域声明一个 final 变量。修饰成员变量时，定义时同时给出初始值，而修饰局部变量时不做要求。这个语句声明了一个 final 变量并对它进行了初始化。如果在后面还想给 piVar 赋其他的值，就会导致编译错误，因为 final 变量的值不能再改变。

(2) final 修饰成员方法，则该方法不能被子类重写。有些方法希望始终保持它在父类中的定义而不会被子类修改以保证安全，此时可以使用 final 关键字来阻止方法重写。例如：

```
public final returnType methodName(paramList){
    …
}
```

(3) final 修饰类，则类不能被继承。由于安全性的原因或者是面向对象的设计上的考虑，有时希望一些类不能被继承，例如，Java 中的 String 类，它对编译器和解释器的正常运行有很重要的作用，不能对它轻易改变，因此把它修饰为 final 类，使它不能被继承，这就保证 String 类型的唯一性。同时，如果认为一个类的定义已经很完美，不需要再生成它的子类，这时也应把它修饰为 final 类，定义一个 final 类的格式如下：

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

如果一个类被声明为 final 的，则这个类中的所有方法也自动成为 final 的。

(4) final 修饰引用类型变量，针对引用类型变量的 final 修饰符也是容易混淆的地方。实际上 final 只是修饰引用变量自身的值，如限定类变量保存的实例地址不能变。至于该变量所引用的对象，内容是否能变，那就管不着了。所以，对于如下语句：

```
final StringBuffer strConst=new StringBuffer();
```

可以修改它指向的对象的内容，如：

```
strConst.append(" ");
```

但是不能修改它的值，如：

```
strConst=null;
```