第5章 Java面向对象高级特性

本章学习目的与要求

本章主要讲解 Java 语言中一些高级特性。通过本章的学习，掌握一些用来修饰类成员的修饰符，抽象方法与抽象类的关系；掌握如何声明和实现一个或多个接口；认识内部类，掌握如何定义内部类，如何访问内部类；掌握自动装箱与拆箱以及枚举、注解和 Lambda 表达式等。

本章主要内容

本章主要介绍以下内容：

- 静态变量、静态方法。
- 最终类、常量。
- 抽象方法和抽象类。
- 接口。
- 内部类。
- 自动装箱与拆箱。
- 枚举。
- 注解。
- Lambda 表达式。

5.1 静态变量、方法和初始化块

类定义的内容主要包括成员变量、成员方法、构造方法和初始化块等，在定义方法和变量时，可以在之前用关键字 static(静态)修饰，表明它们是属于类的，称为静态方法(类方法)或静态变量(类变量)。

5.1.1 静态变量

用 static 修饰的成员变量即为静态变量(类变量)，若无

static 修饰的成员变量则是实例变量。静态变量或类变量是一种全局变量，它属于某个类，不属于某个对象实例，是在各对象实例之间共享。如果想访问静态变量可以直接通过类名来访问，可以不通过实例化访问它们。实例变量必须通过对象实例来访问它们。

【例 5-1】 静态变量与实例变量的访问。

```
class Example05_1 {
    int a =1;
    static int b =2;
}
class Example05_1Demo {
    public static void main(String args[]) {
        System.out.println("b=" +Example05_1.b);
        Example05_1.b * =2;
        Example05_1 o1 =new Example05_1();
        o1.a =10;
        System.out.println("o1.a="+o1.a);
        System.out.println("b="+o1.b);
        Example05_1 o2=new Example05_1();
        System.out.println("o2.a=" +o2.a);
        System.out.println("b="+o2.b);
    }
}
```

在类 Example05_1 中定义了两个成员变量 a、b，其中变量 b 由 static 修饰，所以变量 b 为静态变量，如果想访问静态变量 b，通过类名来访问 Example05_1.b 即可。如果想访问实例变量 a，就要通过对象实例来访问 o1.a 或 o2.a。静态变量 b 是在各对象实例间共享，而实例变量 a 是每个对象实例都独有的。

程序运行结果：

```
b=2
o1.a=10
b=4
o2.a=1
b=4
```

5.1.2 静态方法

同样，用 static 修饰的成员方法即为静态方法（类方法），调用静态方法可以通过类名来调用，即不用实例化即可调用它们。

【例 5-2】 静态方法的调用。

```
class Example05_2{
    public static int add(int x,int y){
```

```
        return x+y;
    }
}
class Example05_2Test{
    public void method(){
      int a=1;
      int b=2;
      int c=Example05_2.add(a,b);
    }
}
```

add(int x,int y)方法是一个用 static 修饰的静态方法,所以可以通过类名来调用 Example05_2.add(a,b)。

成员变量可以分为实例变量和静态变量两种;同样,方法也可以分为实例方法和静态方法两种。在方法使用变量时,需要注意以下规则:

- 实例变量或者类变量在定义初始化时,都不能超前引用。
- 实例方法既可以使用实例变量,又可以使用静态变量;而静态方法只能使用静态变量,不能直接使用实例变量。

【例 5-3】 以下实例使用了不正确的操作。

```
class Example05_3{
    String str="hello";
    public static void main(String args[])    {
        System.out.println(str);
    }
}
```

编译时错误信息如下:

```
nonstatic variable str cannot be referenced from a static context "System.out.
println(str);"
```

为什么不正确?因为静态方法不能直接使用实例变量。

解决的方法:

方法一:将变量改成静态变量。例如:

```
static String str="hello";
```

方法二:先创建一个类的实例,在用该实例调用实例变量。例如:

```
System.out.println(new Example05_3().str);
```

main()方法也是一个静态方法,为什么要把一个 main()方法定义为一个静态方法?执行一个 Java 程序的时候,Java 都是以类作为程序的组织单位,就是由类组成的,那么要

执行的时候，并不知道这个 main()方法要放到哪一个类当中，也不知道是否需要产生这个类的对象实例，为了解决程序的运行问题，就将 main()方法定义为是静态方法，当加载所含 main()方法的类时，main()方法就被加载了，不需要产生该类的对象；否则，如果 main()方法被定义一个成员方法或非静态方法，那必须要产生一个类对象之后，才能调用 main()方法，当然程序也就没有办法运行起来了。

5.1.3 静态初始化块

类变量的初始化也可以通过静态初始化块来进行。静态初始化块是一个块语句，代码放置在一对大括号内，大括号前用关键字 static 修饰：

```
static {…}
```

一个类中可以定义一个或多个静态初始化块。静态初始化块会在加载类时调用而且只被调用一次。

【例 5-4】 静态初始化块。

```
class Example05_4  {
    static int i =5;
    static int j =6;
    static int k;
    static void method() {
        System.out.println("k="+k);
    }
    static  {
        if(i * 5 >=j * 4) k =10;
    }
    public static void main(String args[]) {
      Example05_4.method();
    }
}
```

程序运行结果：

```
k=10
```

method()方法是静态方法，所以最后一行代码 Example05_4.method()通过类名就可以调用该方法，另外，类里定义的静态变量 i、j、k 和第 8 行的静态初始化块在类载入时就已经建立和执行初始化了。

5.2 最终类、变量和方法

final(最终)也是一个重要的关键字，它可以用来修饰类、变量和方法。其中：

- 如果 final 在类之前，则表示该类不能被继承，即 final 类不能作为父类，任何 final

类中的方法自动成为 final 方法，一个类不能同时用 abstract 和 final 修饰。

- 如果 final 在方法之前，则防止该方法被覆盖，final 方法不能被重写，因此，如果在子类中有一个同样签名的方法，那么将得到一个编译时错误。
- 如果 final 在变量之前，则定义一个常量，一个 final 变量的值不能被改变，并且必须在一定的时刻赋值。

【例 5-5】 用 final 修饰类，打印字符串“amethod”。

```
final class Example05_5{
  public void amethod(){
      System.out.println("amethod");
  }
}
class Fin {
  public static void main(String[] args){
      Example05_5 b=new Example05_5();
      b.amethod();
  }
}
```

程序运行结果：

```
amethod
```

5.3 抽象方法与抽象类

Java 中可以定义一些不含方法体的方法，它的方法体的实现交给该类的子类根据自己的情况去实现，这样的方法就是 abstract 修饰符修饰的抽象方法。包含抽象方法的类就称为抽象类，也要用 abstract 修饰符修饰。

5.3.1 抽象方法

用 abstract 来修饰一个方法时，该方法即为抽象方法。形式如下：

```
abstract [修饰符] <返回类型>  方法名称([参数表]);
```

抽象方法并不提供实现，即方法名称后面只有小括号而没有大括号的方法实现。包含抽象方法的类必须声明为抽象类，抽象超类的所有具体子类都必须为超类的抽象方法提供具体实现。

5.3.2 抽象类

使用关键字 abstract 声明抽象类，形如：

```
[public] abstract class 类名
```

抽象类通常包含一个或多个抽象方法(静态方法不能为抽象方法)。

说明:

- 抽象类必须被继承,抽象方法必须被重写。
- 抽象类不能被直接实例化。因此,它一般作为其他类的超类,使用抽象超类来声明变量,用以保存抽象类所派生的任何具体类的对象的引用。程序通常使用这种变量来多态地操作子类对象。与 final 类正好相反。
- 抽象方法只须声明,无须实现。定义了抽象方法的类必须是用 abstract 修饰的抽象类。

如果声明如下一个类:

```
class Shapes{
    abstract double getArea();
    void showArea(){
      System.out.println("Area="+getArea());
    }
}
```

那么在编译的时候就会报错,因为在 Shape 类中定义了两个方法,其中 getArea()方法是用 abstract 修饰的抽象方法,showArea()方法是成员方法,只要在类定义中出现至少一个抽象方法,那么该类也必须要定义成抽象类,所以要把 Shape 类也声明成抽象的,即第 1 行修改为

```
abstract class Shapes{}
```

5.3.3 扩展抽象类

抽象类不能被直接实例化,其目的是提供一个合适的超类,以派生其他类。

用于实例化对象的类称为具体类,这种类实现它们声明的所有方法。抽象超类是一般类,可以被看作是对其所有子类的共同行为的描述,并不创建出真实的对象。在创建对象之前,需要更为专业化的类,所以要得到抽象超类的任何一个具体类(非抽象子类),就必须提供该抽象类中的所有抽象方法的实现。例如:

【例 5-6】 扩展抽象类 Example05_6 使用如下。

```
abstract class A
{
    int x;
    abstract int m1();
    abstract int m2();
}
abstract class B extends A
{
    int y;
    int m1()
    {
```

```
        return x+y;
    }
}
class C extends B
{
    int z;
    int m2()
    {
        return x+y+z;
    }
}
```

子类 B 和 C 会继承其父类中的所有方法(包括成员方法和抽象方法),那么这个子类只有覆盖实现其父类中的所有抽象(abstract)方法才能被定义成非抽象类(如子类 C),否则也只能被定义成抽象类(如子类 B)。

5.4 接口

从本质上讲,接口(interface)是一种特殊的抽象类,这种抽象类中只包含常量和方法的定义,而没有方法的实现。那么,为什么要使用接口呢? 这是因为以下几个原因:

- 通过接口可以实现不相关类的相同行为,而无须考虑这些类之间的层次关系。
- 通过接口可以指明多个类需要实现的方法。
- 通过接口可以了解对象的交互界面,而无须了解对象所对应的类。

5.4.1 接口的定义

在 Java 中,接口是由一些常量和抽象方法所组成的,接口中也只能包含这两种要素,一个接口的声明跟类的声明是一样的,只不过把关键字 class 换成了 interface,接口定义的一般格式如下:

```
[public] interface<接口名>[extends<直接超接口名表>]{
<类型><有名常量名>=<初始化表达式>;…
<返回类型><方法名>(<行参表>);…
}
```

有 public 修饰的接口,能被任何包中接口或类访问;没有 public 修饰的接口,只能在所在包内被访问。

接口中的方法不提供具体的实现,其方法体用分号代替。其中,接口中的变量必须是用 public static final 修饰的,接口中的方法必须是用 public static 修饰的。如果不使用这些修饰符,Java 编译器会自动加上。

【例 5-7】 接口定义示例。

```
interface A{
    void method1(int i);
```

```
    void method2(int j);
}
```

在接口 A 中定义了两个方法，这两个方法虽然只有返回类型，但是系统会自动为这两个方法加上 public static 修饰符。

5.4.2 接口的实现

有了接口之后，任何想要拥有接口所定义的类，就必须去实现这个接口。继承类是用 extends 关键字，而实现接口则使用 implements 关键字。在子类中可以使用接口中定义的常量，而且必须实现接口中定义的所有方法，否则子类就变为抽象类了。

下面是一个实现例 5-7 中接口 A 的实现类：

```
class B implements A{
    public void method1(int i){ …}
    public void method2(int j){ …}
}
```

利用接口可以实现多重继承，即一个类可以实现多个接口，在 implements 子句中用逗号分隔。

【例 5-8】 一个类在继承同时实现多个接口。

```
interface Sittable{
    void sit();
}
interface Lie{
    void sleep();
}
interface HealthCare{
    void massage();
}
class Chair implements Sittable{
    public void sit(){};
}
/* interface Sofa extends Sittable,Lie          //接口可以实现多重继承,用逗号相隔
{
}*/
class Sofa extends Chair implements Lie,HealthCare
                                    //一个类既可从父类中继承,同时又可实现多个接口
{
    public void sleep(){};
    public void massage(){};
}
```

上面例 5-8 中定义了 3 个接口 Sittable、Lie 和 HealthCare。定义了 Chair 类来实现 Sittable 接口；Sofa 类继承自 Chair 接口，又实现了 Lie 和 HealthCare 接口。

接口最主要的功能是让不同实现的类拥有相同的访问方式，用户不需要知道具体是用什么方式实现的，只要知道这个接口提供了那些方法即可。

说明：自 JDK8 以后，可以使用默认方法和静态方法这两个概念来扩展接口的声明。使用 default 关键字修饰默认方法，使用 static 关键字修饰静态方法，并提供默认的实现。接口的默认方法和静态方法的引入，其实可以认为是引入了 C++ 中抽象类的理念，这样有利于代码的可复用性。

JDK8 中能够在接口中定义带方法体的默认方法和静态方法，这样会导致一个问题：当两个默认方法或者静态方法中包含一段相同的实现逻辑时，程序必然考虑将这段实现逻辑抽取成工具方法，而工具方法应该是隐藏的。所以，JDK9 更新弥补了这一缺陷，增加了可带方法体的私有方法。

【例 5-9】 用默认方法和静态方法拓展接口的声明。

```
package sample;
interface DefaultImplementation {
    default void implementmethod() {
        System.out.println("default implementation");
    }
    static void staticmethod() {
        System.out.println("static implemetation");
    }
    class DefaultableImpl implements DefaultImplementation {
    }
    class OverridableImpl implements DefaultImplementation {
        public void implementmethod() {
            System.out.println("override default implementation");
        }
        static void staticmethod(){
            System.out.println("override static implemetation");
        }
    }
    public static void main(String[] args) {
        DefaultableImpl dimpl =new DefaultableImpl();
        OverridableImpl oimpl =new OverridableImpl();
        dimpl.implementmethod();
        oimpl.implementmethod();
        oimpl.privatemethod();
    }
}
```

程序运行结果：

```
default implementation
override default implementation
override static implemetation
```

5.5 内部类

嵌套在另一个类中定义的类就是内部类(Inner Classes)。包含内部类的类称为外部类。

5.5.1 认识内部类

内部类在 Java 最初的 1.0 版本中是不允许的，在后面的 Java 1.1 版本中才添加了内部类。内部类产生的原因主要是图形用户界面程序中的事件处理(将在后面详细讲解对于某些类型的事件内部类如何被用来简化代码)。它的存在的主要有两个目的：

(1) 可以让程序设计中逻辑上相关的类结合在一起。有些类必须要伴随着另一个类的存在才有意义，如果两者分开，则可能在类的管理上比较麻烦，所以可以把这样的类写成一个 Inner Class。

(2) Inner Class 可以直接访问外部类的成员。因为 Inner Class 在外部类中，可以访问任何的成员，包括声明为 private 的成员。

下面的程序定义了一个内部类。创建内部类的基本语法如下。

【例 5-10】 内部类的声明。

```
class Outer                          //这是一个外部类
{
  int outer_x =100;                  //外部类成员
  class Inner                        //这是一个内部类
  {
      void display()                 //内部类成员
      {
        System.out.println("display: outer_x =" +outer_x);
      }
  }
}
```

在本程序中，类 Inner 嵌套在另一个 Outer 类中定义，类 Inner 就是内部类(Inner Classes)。包含内部类的 Outer 类就是外部类。

按照 Inner Class 声明的位置，可以把它分成两种：一种是类成员式的，就像属性、方法一样，把一个类声明为另一个类的成员；另一种是区域式的，也就是把类声明在一个方法中。由这两种方式所派生出来的，按它的存在范围又分成 4 种级别，第一种是对象级别，第