高级类特性

抽象类和接口是面向对象程序设计的两种重要机制，与类相比较是更高层次的抽象，两者本质上具有相似之处。Java 中引入内部类使得程序更加简洁，便于以类的组织来划分程序的层次结构和类的命名，内部类的功能完全可以使用普通类来替代实现。Lambda 表达式是 Java 8 中重要的新特性，支持将代码作为方法参数，可以使用更加简洁的代码来创建只有一个抽象方法的接口实例。反射是 Java 语言特有的一种机制，即通过它可以了解任意类具有的属性、构造方法和普通方法。

本章主要介绍 Java 中的抽象类、接口、内部类、Lambda 表达式和反射机制。

5.1 抽 象 类

在面向对象程序设计中，所有对象都是通过类来描述的，但不是所有类都是用来描述对象的。如果一个类中没有包含足够的信息来描述一个具体的对象，这样的类就是抽象类。抽象类往往用来表征问题领域中分析、设计得出的抽象概念，是对一系列看上去不同，但是本质上相同的具体概念的抽象。假如要开发一个动物管理软件，设计动物类时发现无法具体描述它们的行为。动物类包括人类、兽类、鸟类和鱼类等，它们并不相同但又都属于动物。人类、兽类、鸟类和鱼类都具有各自的寿命和吃食物、运动的行为，但是它们吃食物、运动的方式方法（实现细节）并不相同。此时只能去定义比类更抽象的抽象类来描述动物类，动物类中只定义动物具有的行为方法，不提供方法的具体实现。

【例 5-1】 动物类定义。

程序 Animal.java 如下。

```
public abstract class Animal{
    public int age;
    public abstract void eat();
    public abstract void move();
}
```

程序 Animal.java 中定义的动物类 Animal 就是一个抽象类，通过该例先来了解一个抽象类的定义，下面详细介绍抽象类。

Java 中关于抽象类的使用规则如下。

(1) abstract 修饰符可以修饰类和方法,用 abstract 修饰的类称为抽象类,用 abstract 修饰的方法称为抽象方法,抽象方法属于一种不完整的方法,只含有一个声明,没有方法体(具体实现细节)。抽象方法必须使用 abstract 修饰符修饰。例如,动物类 Animal 中的"public abstract void eat();"方法,它和"public void eat(){}"方法具有本质的区别,"public void eat(){}"是一个普通方法,具有方法体,只不过具体实现细节为空实现而已。

(2) 抽象类不能实例化对象,只能声明引用变量。抽象类没有包含足够的信息来描述一个具体的对象,所以不能实例化对象,但是抽象类可以声明引用。

```
Animal animal;                    //合法
new Animal();                     //非法
Animal animal =new Animal();      //非法
```

(3) 含有抽象方法的类必须声明为抽象类,抽象类可以不含有抽象方法。修改【例 5-1】如下。

```
public class Animal{
    public int age;
    public abstract void eat();
    public abstract void move();
}
```

上述代码编译时出错,Animal 类中含有抽象方法 eat()和 move(),声明 Animal 类时必须用 abstract 修饰。

修改【例 5-1】如下。

```
public abstract class Animal{
    public int age;
    public void eat(){
        System.out.println("Animal eating!");
    }
    public void move(){
        System.out.println("Animal moving! ");
    }
}
```

从 Java 语法角度来讲,上述代码是正确的。在 Java GUI 应用程序开发中,事件处理的适配器类都是抽象类,但这些抽象类中其实并不存在抽象方法。之所以定义为抽象类,就是要避免开发者使用适配器类实例化对象。

(4) 定义抽象类的主要目的是在定义新类时继承该抽象类。通过继承,由其子类来发挥作用,抽象类实现了代码的重用和规划作用。当子类继承抽象类时,必须实现抽象类中的所有抽象方法,否则子类也要声明为抽象类。

【例 5-2】 定义 Bird 类继承 Animal 类(【例 5-1】),在 Bird 类中实现类 Animal 两个抽象方法 eat()和 move()。

程序 Bird.java 如下。

```
public class Bird extends Animal {
    public void eat() {
        System.out.println("Bird is eating insect!");
    }
    public void move() {
        System.out.println("Bird is flying!");
    }
}
```

【例 5-3】 定义 Fish 类继承 Animal 类,在 Fish 类中实现类 Animal 的两个抽象方法 eat()和 move()。

程序 Fish.java 如下。

```
public class Fish extends Animal {
    public void eat() {
        System.out.println("Big Fish is Small Fish!");
    }
    public void move() {
        System.out.println("Fish is swimming!");
    }
}
```

【例 5-2】和【例 5-3】定义了 Bird 类和 Fish 类来继承 Animal 类,分别根据鸟类 Bird 和鱼类 Fish 的吃食物和行动的行为特征给出具体的行为实现细节。

【例 5-4】 定义 Person 类继承 Animal 类。

程序 Person.java 如下。

```
public abstract class Person extends Animal {
    public abstract void eat();
    public void move() {
        System.out.println("Person is running!");
    }
}
```

在【例 5-4】中,在 Person 类中实现类 Animal 中的两个抽象方法 eat()和 move()中的 move()方法,eat()方法并没有实现,那么 Person 类必须使用 abstract 修饰符声明为抽象的,否则编译出错。

(5) 抽象类及抽象方法不能被 final 修饰符修饰。抽象类和抽象方法利用 final 来修饰意味着抽象类不能被继承,抽象方法不能被重写(覆盖),那么抽象类和抽象方法就失去了存在的意义。

【例 5-5】 定义 Person1 类继承 Animal 类,并声明为 final。

程序 Person1.java 如下。

```
public final abstract class Person1 extends Animal {
    public final abstract void eat();
```

```
    public void move() {
        System.out.println("Person is running!");
    }
}
```

【例 5-5】的程序 Person1.java 编译出错。

(6) 抽象类的使用方式为：利用抽象父类声明引用变量，并让其指向子类的对象(产生多态)。

【例 5-6】 定义类 AnimalTest 使用 Animal、Bird 和 Fish 类来演示多态的使用。

程序 AnimalTest.java 如下。

```
public class AnimalTest {
    public static void main(String[] args) {
        Animal a1 =new Bird();      //父类 Animal 的引用指向子类 Bird 的对象
        Animal a2 =new Fish();      //父类 Animal 的引用指向子类 Fish 的对象
        a1.age=3;
        a2.age=2;
        a1.eat();
        a2.move();
    }
}
```

程序运行结果如下。

```
Bird is eating insect!
Fish is swimming!
```

5.2 接　　口

在 Java 中，除了类和数组之外，接口也是引用类型之一。Java 中的接口和抽象类在本质上是相似的，只是接口比抽象类更抽象。

5.2.1 接口概念

学习 Java 接口之前，应先了解计算机接口的概念。计算机接口是一套规范，满足这个规范的设备就可以组装到一起，从而实现该设备的功能。如计算机主板上的显卡接口。在计算机软件中，同一计算机不同功能层之间的通信规则称为接口，也可以说是对规则进行定义的引用类型。在 Java 语言中，接口同样是一种规范和标准，用于约束类的行为。如在 Java 中的一个类实现 java.lang.Comparable 接口，就必须实现该接口中的抽象方法 CompareTo()，以此来规定类中必须要实现的方法。

这里举一个例子，以便更好地理解接口的概念。Java 数据库连接技术即 JDBC(Java DataBase Connection)，使得 Java 可以操作 Oracle、DB2、MySQL 等各种不同类型的数据库，但需要各数据库厂商提供数据库驱动程序。在 Java 语言访问不同数据库的实现过程

中,接口发挥了巨大作用。其实现过程是:首先甲骨文(Oracle)公司制定数据库访问的接口(即一系列相关规范),各个数据库厂商实现该接口(即数据库驱动程序)。程序在操作不同厂商的数据库时,只要加载对应数据库的驱动程序,其他的操作代码都一样。后面的接口应用中,DriverTest 实例模拟这个实现过程。

Java 中接口的定义包括两个方面:一是 Java 语言中存在的结构,有特定的语法和结构;二是一个类所具有的方法特征集合,是一种逻辑上的抽象。Java 中的接口是一系列方法的声明,是一些方法特征的集合。一个接口只有方法的特征,没有方法的实现,因此这些方法可以在不同的地方被不同的类实现,而这些实现可以具有不同的行为。

5.2.2 接口定义

Java 中接口定义语法规则如下。

```
[访问修饰符] interface 接口名 [extends 父接口列表]{
    [public][static][final] 常量名;
    [public][abstract]<方法返回类型>方法名(参数列表);
    ...//default 方法和 static 方法
}
```

从接口的定义语法规则中可以看出:定义接口使用关键字 interface,接口是抽象方法和常量值定义的集合。从本质上讲,接口是一种特殊的抽象类,包含常量和抽象方法的定义。

【例 5-7】 定义接口 Runner。

程序 Runner.java 如下。

```
public interface Runner{
    public static final int id=1;
    public abstract void start();
    public abstract void run();
    public abstract void stop();
}
```

定义接口的主要目的是让不同的类来实现,使接口起到桥梁的作用,那么接口的成员都是公有的。结合上述接口定义语法规则可以得出:接口中定义的属性必须是 public static final 的,方法必须是 public abstract 的,因此这些修饰符可以部分或全部省略。接口 Runner 也可以做如下修改。

```
public interface Runner{
    int id=1;
    void start();
    void run();
    void stop();
}
```

上述修改后的接口 Runner 定义和例 5-7 中的 Runner 接口定义相比是完全等价的。

5.2.3 接口的默认方法和静态方法

在 Java 8 以前，接口中只能有抽象方法（public abstract，修饰的方法）和全局静态常量（public static final，常量）。但是在 Java 8 中，允许接口中包含具有具体实现的方法，称为“默认方法”，默认方法使用 default 关键字修饰。Java 8 中，接口还允许添加静态方法，静态方法就是类方法，需要在方法前使用 static 关键字修饰。

Java 8 之前的接口可以很好地实现面向抽象而不是面向具体编程，但当需要修改接口时，则需要修改全部实现该接口的类。在 Java 8 接口中，增加默认方法和静态方法的目的是扩展接口的功能，让接口在发布后仍能继续演化，而不影响所有实现接口类的使用。

【例 5-8】 接口中定义默认方法和静态方法。

程序 Runner.java 如下。

```
public interface Runner {
    int id=1;
    void start();
    void run();
    void stop();
    default void print() {
        System.out.println("我是一个 runner");
    }
    static String msg() {
        return "接口中的静态方法";
    }
}
```

【例 5-8】中的 print()方法为默认方法，使用 default 修饰，方法有方法体。默认方法不能通过接口直接访问，必须通过接口实现类的实例访问。【例 5-8】中的 msg()方法为静态方法，使用 static 修饰，静态方法具有类共享的特性，可以直接通过接口名访问，也可以通过接口实现类的实例访问。

5.2.4 接口的多继承

和 Java 中类的继承关系一样，接口之间也可以继承，即定义接口时可以继承已有的接口，添加新的常量和抽象方法定义，在父接口的基础上进行下一步扩展。当然，最终发挥作用的还是接口的实现类，只不过接口之间的继承支持多重继承，即一个接口可以同时继承一个或多个已有的接口。

【例 5-9】 演示接口的多继承使用。

程序 D.java 如下。

```
interface A{
    public void ma();
}
```

```
interface B{
    public void mb(int i);
}
interface C extends A,B{              //接口 C 同时继承接口 A 和 B
    public void mc();
}
public class D implements C{
    public void ma(){
        System.out.println("Implements methos ma");
    }
    public void mb(int i){
        System.out.println(2000+i);
    }
    public void mc(){
        System.out.println("Hello!");
    }
    public static void main(String[] args){
        D d=new D();
        d.ma();
        d.mb(100);
        d.mc();
    }
}
```

运行结果如下。

```
Implements methos ma
2100
Hello!
```

5.2.5 接口实现

接口和抽象类本质上是相似的,都是抽象的概念,只是接口比抽象类更抽象,无法去具体描述一个对象,所以接口和抽象类一样不能实例化对象,只能声明引用变量。

```
Runner r;                             //合法
new Runner();                         //非法
Runner r =new Runner();               //非法
```

接口定义的是一套行为规范,定义接口的主要目的是让不同的类来实现。一个类实现某个接口就要遵守接口中定义的规范,也就是要实现接口中定义的所有抽象方法。类实现接口(类的定义)的语法规则如下。

```
[访问修饰符][非访问修饰符] class 类名 [extends 超类名称][implements interface1,
interface2,…,interfacen]{
        属性声明;
```

```
        构造方法声明;
        方法声明及方法体;
}
```

实现接口时要注意以下规则。

(1) 类实现接口使用关键字 implements 来声明。

(2) 一个类可以同时实现多个接口,接口间使用逗号进行间隔。

(3) 一个类在实现一个或多个接口时,必须实现这些接口中所有的抽象方法,否则该类须声明为抽象类。

【例 5-10】 定义一个 Person 类,使其实现 Runner 接口。

程序 Person.java 如下。

```
public class Person implements Runner {
    //实现抽象方法 start()
    public void start() {
        System.out.println(id);
        System.out.println("Person is Start Running!");
    }
    //实现抽象方法 run()
    public void run() {
        System.out.println("Person is Running!");
    }
    //实现抽象方法 stop()
    public void stop() {
        System.out.println("Person is Stop Running!");
    }
    //可以重写接口中的默认方法
    public void print() {
        System.out.println("我是一个 Person");
    }
    //定义一个实现类自己的方法
    public void work() {
        System.out.println("Person 需要工作");
    }
}
```

Person 类就是 Runner 接口的实现类,它实现了接口 Runner 中的所有抽象方法,并重写了接口中的默认方法,但是默认方法不是必须要重写的。

修改【例 5-10】,代码如下。

```
public abstract class Person implements Runner {
    public void start() {
        System.out.println(id);
        System.out.println("Person is Start Running!");
    }
}
```

修改后的 Person 类只实现了 Runner 接口中的抽象方法 start(),没有实现接口 Runner 中的其他抽象方法,因而 Person 类必须用 abstract 修饰,否则编译出错。

由此得出结论:一个类实现一个接口,必须实现接口中所有的抽象方法,否则该类必须声明为抽象类。

定义类去实现接口和继承已有的类本质是相似的,可以"继承"接口中常量和抽象方法。接口虽然只能声明接口引用,不能实例化对象,但是接口引用可以指向实现类的对象,这一点和父类引用指向子类对象情况一样。

【例 5-11】 演示接口的使用。

程序 RunnerTest.java 如下。

```
public class RunnerTest {
    public static void main(String[] args) {
        Runner r=new Person();      //接口 Runner 的引用 r 指向实现类 Person 的对象
        //通过接口实现类对象调用抽象方法
        r.start();
        r.run();
        r.stop();
        //通过接口实现类对象调用默认方法
        r.print();
        //接口中静态方法的调用
        Runner.msg();
        //r.work(); 编译错误
    }
}
```

Runner 是一个接口,没有办法实例化对象,在上述代码中,用 Runner 接口的实现类 Person 类来实例化对象,并将对象返回给 Runner 类型的引用变量 *r*,通过 *r* 来调用实现类中实现的抽象方法以及重写的默认方法。对于接口的静态方法,通过接口名进行直接访问。和继承中父类引用不能调用子类中独有的方法一样,接口引用也不能调用实现类中独有的方法,因此 r.work()会出现编译错误。

程序运行结果如下。

```
1
Person is Start Running!
Person is Running!
Person is Stop Running!
我是一个 Person
```

5.2.6 接口的多重实现

Java 只支持类的单继承,不支持多继承,即定义类时只能继承一个已有类,不能同时继承多个已有类。Java 中的类实现接口和类继承不同,在定义一个类时可以同时实现一

个或多个接口。

【例 5-12】 演示接口的多重实现使用。

程序 Test.java 如下。

```
interface Swimmer {
    public abstract void swim();
}
interface Runner{
    void run();
}
abstract class Animal{
    public abstract void eat();
}
class Person extends Animal implements Runner,Swimmer{
//类 Person 继承 Animal 类的同时实现了接口 Runner 和 Swimmer
    public void eat() {
        System.out.println("I am eatting!");
    }
    public void run() {
        System.out.println("I am Running!");
    }
    public void swim() {
        System.out.println("I am Swimming!");
    }
}
public class Test{
    public static void main(String[] args){
        Test t=new Test();
        Person p=new Person();
        t.m1(p);
        t.m2(p);
        t.m3(p);
    }
    public void m1(Runner r){
        r.run();
    }
    public void m2(Swimmer s){
        s.swim();
    }
    public void m3(Animal a){
        a.eat();
    }
}
```

上述程序 Test.java 代码中的 Test 类也可以改为下述代码，道理是完全一样的。