第5章

Java 高级特征

在第 4 章 Java 面向对象特征的基础上，本章将进一步介绍 Java 的高级面向对象特征，其中某些特征如接口是 Java 独有的语言机制。本章介绍的具体内容包括通过 static 关键字定义的类变量、类方法和初始化程序块，final 关键字，抽象类，接口(interface)，package，泛型与集合类，枚举类型，包装类与自动装箱和拆箱等。

5.1 static 关键字

static 关键字可以用来修饰类的成员变量、成员方法和内部类，使得这些类成员的创建和使用，与类相关而与类的具体实例不相关，因此以 static 修饰的变量或方法又称为类变量和类方法。

5.1.1 类变量/静态变量

在成员变量声明时使用 static，则该变量就称为类变量或静态变量。静态变量只在系统加载其所在类时分配空间并初始化，并且在创建该类的实例时将不再分配空间，所有的实例将共享类的静态变量。因此静态变量可用来在实例之间进行通信或跟踪该类实例的数目。

例 5-1 定义了一个类 Count。Count 中定义了一个静态变量 counter。创建 Count 对象时，将递增 counter，并把 counter 的值赋予该对象的 serialNumber 变量。这样，如果把 serialNumber 看作对象的序列号，则通过静态变量 Counter 将使 Count 类的每个对象都被赋予了唯一的序列号，这些序列号从 1 开始递增。

例 5-1 为 Count 类的对象赋予递增的序列号。

```
class Count{
    private int serialNumber;
    public static int counter = 0;
    public Count(){

        counter++;
        serialNumber = counter;
        }
    public int getSerialNumber(){
        return serialNumber;
    }
}
public class TestStaticVar{
```

```
    public static void main(String[] args){
      Count[] cc = new Count[10];
      for(int i = 0;i < cc.length;i++){
          cc[i] = new Count();
          System.out.println("cc[" + i + "].serialNumber = " + cc[i].getSerialNumber());
      }
    }
}
```

例 5-1 的运行结果如下：

```
cc[0].serialNumber = 1
cc[1].serialNumber = 2
cc[2].serialNumber = 3
cc[3].serialNumber = 4
cc[4].serialNumber = 5
cc[5].serialNumber = 6
cc[6].serialNumber = 7
cc[7].serialNumber = 8
cc[8].serialNumber = 9
cc[9].serialNumber = 10
```

Java 中没有全局变量，但静态变量是在一个类的所有实例对象中都可以访问的变量，有点类似于其他语言中的全局变量。

静态变量只依附于类，而与类的实例对象无关，所以对于不是 private 类型的静态变量，可以在该类外直接用类名调用，而不像实例变量那样需要通过实例对象才能访问。例如，可以在下面的类中直接对例 5-1 中的 Count 类的静态变量 Counter 进行访问。

```
public class OtherClass{
    public void incrementNumber(){
        Count.Counter++;
    }
}
```

5.1.2　类方法/静态方法

在类的成员方法声明中带有 static 关键字，则该方法就称为类方法或静态方法。静态方法要通过类名而不是通过实例对象访问。在例 5-2 中，类 GeneralFunction 定义了一个实现两个整数加法的静态方法，在另一个类 UseGeneral 中可以通过 GeneralFunction 类直接访问。

例 5-2　对 GeneralFunction 类静态方法的访问。

```
class GeneralFunction{
    public static int add(int x, int y){
        return x + y;
    }
}
public class UseGeneral{
    public static void main(String[] args){
```

```
        int c = GeneralFunction.add(9,10);
        System.out.println("9 + 10 = " + c);
    }
}
```

例 5-2 的运行结果为:

```
9 + 10 = 19
```

在静态方法的编写与使用时应该注意下列问题。

(1) 因为静态方法的调用不是通过实例对象进行的,所以在静态方法中没有 this 指针,不能访问所属类的非静态变量和方法,只能访问方法体内定义的局部变量、自己的参数和静态变量。

(2) 子类不能重写父类的静态方法,但在子类中可以声明与父类静态方法相同的方法,从而将父类的静态方法隐藏。另外子类不能把父类的非静态方法重写为静态的。例如,下列代码将出现编译错误。

```
class ClassA {
    public void methodOne(int i) {
    }
    public void methodTwo(int i) {
    }
    public static void methodThree(int i) {
    }
    public static void methodFour(int i) {
    }
}

class ClassB extends ClassA {
    public static void methodOne(int i) {
    }//错误!将 ClassA 中的 methodOne()变成静态的
    public void methodTwo(int i) {
    }
    public void methodThree(int i) {
    }//错误!不能重写 ClassA 中的静态方法 methodThree()
    public static void methodFour(int i) {
    }//正确!将把 ClassA 中的 methodFour()方法隐藏
}
```

(3) main()方法是一个静态方法。因为它是程序的入口点,这可以使 JVM 不创建实例对象就可以运行该方法。因此,如果要在 main()方法中访问所在类的成员变量或方法,就必须首先创建相应的实例对象。例如,下面的代码将出现编译时错误。

```
public class wrong{
    int x;
    public static void  x(){
          x = 9;     //错误!访问类的非静态变量
    }
}
```

5.1.3　静态初始化程序

在一个类中，不属于任何方法体并且以 static 关键字修饰的语句块，称为静态语句块。因为静态语句块常用来进行类变量的初始化，所以也称为静态初始化程序块。其定义格式如下：

```
static{
    …
}
```

静态语句块在加载该类时执行且只执行一次。如果一个类中定义了多个静态语句块，则这些语句块将按在类中出现的次序运行。例 5-3 是一个使用静态语句块的例子。

例 5-3　静态语句块与静态变量的访问。

```
class StaticInitDemo{
    static int i;
    static {
        i = 5;
        System.out.println("Static code: i = " + i++ );
    }
}

public class TestStaticInit {
    public static void main(String args[]){
        System.out.println(" Main code: i = " + StaticInitDemo.i);
    }
}
```

例 5-3 的运行结果如下：

```
Static code: i = 5
Main code: i = 6
```

在例 5-3 中，类 StaticInitDemo 定义了类变量 i，并在静态语句块中对 i 赋予了初值 5。由于静态语句块是在类加载时运行，因此系统将首先打印输出 Static code：i＝5，并把 i 的值增加为 6，然后再运行 TestStaticInit 类中的 main()方法，打印输出 Main code：i＝6。

5.2　final 关键字

1. 在类的声明中使用 final

Java 允许在类的声明中使用 final 关键字。被定义成 final 的类不能再派生子类。例如 java.lang.String 类就是一个 final 类。这保证对 String 对象方法的调用确实运行的是 String 类的方法，而不是经其子类重写后的方法。

2. 在成员方法声明中使用 final

对于类中的成员方法也可以定义为 final。被定义成 final 的方法不能被重写。当方法的实现不能被改变，或者方法对于保证对象状态的一致性很关键时，应该把该方法定义为

final。

定义为 final 的方法可以使运行时的效率优化。正如在第 4 章中提到的,对于 final 方法,编译器可以产生直接调用方法的代码,从而阻止运行时刻对方法调用的动态联编。实际上,如果方法被定义为 static 或 private,编译器也将对它们进行上述优化。

3. 在成员变量的声明中使用 final

如果类的成员变量被定义成 final,则变量一经赋值就不能改变,所以可以通过声明 final 变量并同时赋初值来定义常量,并且变量名一般大写。例如:

```
final int NUMBER = 100;
```

如果在程序中要改变 final 变量的值,则将产生编译时错误。如果类的 final 变量在声明时没有赋初值,则在所属类的每个构造方法中都必须对该变量赋值。如果未赋初值的 final 变量是局部变量,则可以在所属方法体的任何位置对其赋值,但只能赋一次值。例 5-4 是对 final 变量声明与赋值的例子。

例 5-4 声明类的 final 变量并在构造方法中赋值。

```
class Customer{
    private final long customerID;
    private static long counter = 200901;
    public Customer(){
        customerID = counter++ ;
    }
    public long getID(){
        return customerID;
    }
    public static void main(String[ ] args){
        Customer[ ] cc = new Customer[5];
        for (int i = 0; i < cc.length; i++ ){
            cc[i] = new Customer();
            System.out.println("The customerID is " + cc[i].getID());
        }
    }
}
```

例 5-4 的运行结果如下:

```
The customerID is 200901
The customerID is 200902
The customerID is 200903
The customerID is 200904
The customerID is 200905
```

5.3 抽 象 类

5.3.1 什么是抽象类

Java 允许在类中只声明方法而不提供方法的实现。这种只有声明而没有方法体的方

法称为抽象方法，而包含一个或多个抽象方法的类称为抽象类。抽象类必须在声明中加 abstract 关键字，而抽象方法在声明中也要加上 abstract。抽象类也可有构造方法、普通的成员变量或方法，也可以派生抽象类的子类。

抽象类在使用上有特殊的限制，即不能创建抽象类的实例。正是为了阻止程序员创建抽象类的实例对象，使编译器在编译时刻对此进行检查，Java 中要将抽象类和抽象方法带上 abstract 标记。如果抽象类的子类实现了抽象方法，则可以创建该子类的实例对象，否则该子类也是抽象类，也不能创建实例。一般将抽象类构造方法的访问权限声明为 protected 而不是 public，从而保证构造方法能够由子类调用而不被其他无关的类调用。例如：

```
abstract class Employee {
  abstract void raiseSalary (int i);
}
class Manager extends Employee {
    void raiseSalary (int i){ … }
}
…
Employee e = new Manager();     //创建 Employee 子类 Manager 的对象
Employee e = new Employee();    //错误!Employee 为抽象类
…
```

5.3.2　抽象类的作用

类是现实世界同类对象的抽象，是 Java 程序中创建对象的模板。抽象类不能实例化对象，那么抽象类的意义是什么呢？程序中定义抽象类的目的是为一类对象建立抽象的模型，在同类对象所对应的类体系中，抽象类往往在顶层。这一方面使类的设计变得清晰，另一方面抽象类也为类的体系提供通用的接口。这些通用的接口反映了一类对象的共同特征。定义了这样的抽象类后，就可以利用 Java 的多态机制，通过抽象类中的通用接口处理类体系中的所有类。

在第 4 章介绍运行时多态的概念时，曾在图 4-9 中给出了一个关于几何形状 Shape 及其子类的例子。在这个类的层次结构中，Shape 类是顶层类。实际上 Shape 类的对象是没有实际意义的。定义 Shape 类的目的并不是为了在程序中创建并操作它的对象，而是为了定义几何形状类体系的通用接口，如 draw()和 erase()，这些接口在 Shape 类中不需要给出具体实现，而由它的各个子类提供自己的实现。因此 Shape 类可以定义为抽象类，而 draw()和 erase()方法可以定义为抽象方法，如图 5-1 所示。

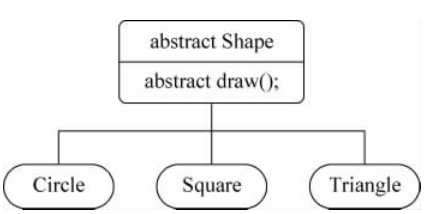

图 5-1　抽象类 Shape 及其类体系

实际上，即使不包括任何抽象方法，也可将一个类声明为抽象类。这样的类往往是没有必要定义任何抽象方法，而设计者又想禁止创建该类的实例对象，此时只需在类的声明中加上 abstract 关键字。

定义抽象类和抽象方法可以向用户和编译器明确表明该类的作用和用法，使类体系设计更加清晰，并能够支持多态，因此是 Java 的一种很有用的面向对象机制。

5.4 接　　口

5.4.1 什么是接口

Java 中的接口(interface)使抽象类的概念更深入一层。接口中声明了方法,但不定义方法体,因此接口只是定义了一组对外的公共接口。与类相比,接口只规定了一个类的基本形式,不涉及任何实现细节。实现一个接口的类将具有接口规定的行为。

在 OOP 中,一个类的"公共接口"可以被认为是使用类的"客户"代码与提供服务类之间的契约或协议。因此可以认为一个接口的整体就是一个行为的协议。实现一个接口的类将具有接口规定的行为,并且外界可以通过这些接口与它通信。有些 OOP 中采用 protocol 关键字,而 Java 中使用 interface 关键字。

下面给出接口的具体定义。

5.4.2 接口的定义

接口的定义包括接口声明和接口体两部分。格式如下:

```
interfaceDeclaration{
    interface Body
}
```

1. 接口声明

接口声明的格式如下:

```
[public] interface InterfaceName [extends listofSuperInterface]{
    …
}
```

其中 public 指明任意类均可以使用这个接口。默认情况下,只与该接口定义在同一个包中的类才可以访问这个接口。extends 子句与类声明中的 extends 子句基本相同,不同的是一个接口可以有多个父接口,用逗号隔开,而一个类只能有一个父类。子接口继承父接口中所有的常量和方法。

2. 接口体

接口体中包含常量定义和方法定义两部分。

在接口中定义的常量默认具有 public,final,static 的属性。常量定义的具体格式为:

```
type NAME = value;
```

其中 type 可以是任意类型,NAME 是常量名,通常用大写,value 是常量值。在接口中定义的常量可以被实现该接口的多个类共享。

在接口中声明的方法默认具有 public 和 abstract 属性。方法定义的格式为:

```
returnType  methodName([paramlist]);
```

接口中只进行方法的声明,而不提供方法的实现。所以,方法定义没有方法体,且以分号";"结尾。另外,如果在子接口中定义了和父接口同名的常量和相同的方法,则父接口中

的常量被隐藏，方法被重写。

注意：接口中的成员不能使用某些修饰符，例如：transient，volatile，synchronized，private，protected。

5.4.3　接口的实现与使用

类的声明中用 implements 子句来表示一个类实现了某个接口，在类体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。一个类可以实现多个接口，在 implements 子句中用逗号分隔。

在类中实现接口所定义的方法时，方法的声明必须与接口中所定义的完全一致。

下面举一个接口及接口实现的例子。现实世界中有很多实体具有飞行的功能。我们可以构造一个公共的接口 Flyer 来抽象描述飞行行为。该接口规定了 3 个方法：起飞、着陆和飞行。接口 Flyer 的定义如下：

```
public interface Flyer{
    public void takeoff();
    public void land();
    public void fly();
}
```

飞机是我们很熟悉的一种具有飞行能力的工具。我们可以定义一个类 Airplane，该类通过实现 Flyer 接口从而对外表征出 Flyer 接口所规定的飞行行为。Airplane 的定义如下：

```
public class Airplane implements Flyer{
    public void takeoff(){
        //加速直至飞起，收起着陆装置等操作
    }
    public void land(){
        //下落着陆装置、减速并降低机翼直到接触地面等操作
    }
    public void fly(){
        //保持所有发动机正常运行等操作
    }
}
```

在程序中，接口可以像类一样作为数据类型来使用，并且可以支持多态。此时任何实现该接口的类都可认为是该接口的“子类”，因此声明为某接口类型的变量，可以指向该接口“子类”的实例，通过这些变量可以访问接口中规定的方法。例 5-5 是通过接口实现多态的例子。

例 5-5 将第 4 章中例 4-18 中的类进行了重定义，将 Shape 定义为描述几何图形的接口，几种图形如 Circle，Square 和 Triangle 都实现 Shape 接口，如图 5-2 所示。

例 5-5 的程序是对例 4-18 的改写。在例 5-5 中将 Shape 定义为接口，Circle，Square 和 Triangle 分别实现了该接口，main()方法实现与例 4-18 相同的操作。

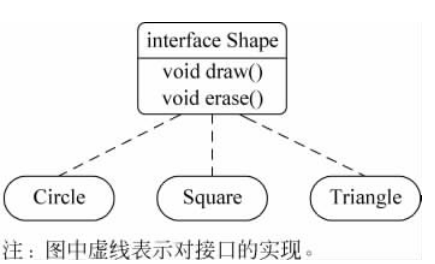

图 5-2　Shape 接口及实现它的各个类

例 5-5 通过接口实现多态示例。

```
import java.util.*;

//将 Shape 定义为 interface
interface Shape {
   void draw();
   void erase();
}

//定义 Circle 类实现 Shape
class Circle implements Shape {
  public void draw() {
    System.out.println("Calling Circle.draw()");
  }
  public void erase() {
    System.out.println("Calling Circle.erase()");
  }
}

//定义 Square 类实现 Shape
class Square implements Shape {
  public void draw() {
    System.out.println("Calling Square.draw()");
  }
  public void erase() {
    System.out.println("Calling Square.erase()");
  }
}

//定义 Triangle 类实现 Shape
class Triangle implements Shape {
   public void draw() {
    System.out.println("Calling Triangle.draw()");
  }
   public void erase() {
    System.out.println("Calling Triangle.erase()");
  }
}

//包含 main()的测试类
public class NewShapes{
  static void drawOneShape(Shape s){
    s.draw();
  }
  static void drawShapes(Shape[] ss){
     for(int i = 0; i < ss.length; i++){
         ss[i].draw();
    }
  }
```

```
public static void main(String[] args) {
    Random rand = new Random();
    Shape[] s = new Shape[9];
    for(int i = 0; i < s.length; i++){
       switch(rand.nextInt(3)) {
              case 0: s[i] = new Circle();break;
              case 1: s[i] = new Square();break;
              case 2: s[i] = new Triangle();break;
       }
    }
    drawShapes(s);
  }
}
```

例 5-5 的某次运行结果如下：

```
Calling Circle.draw()
Calling Triangle.draw()
Calling Square.draw()
Calling Circle.draw()
Calling Triangle.draw()
Calling Triangle.draw()
Calling Triangle.draw()
Calling Square.draw()
Calling Square.draw()
```

由于 Circle，Square 和 Triangle 类的实例是随机生成的，所以例 5-5 各次运行的结果可能不同。注意在例 5-5 中，由于接口 Shape 中声明的方法其访问权限默认是 public，所以在实现 Shape 接口的各类如 Circle，Square 和 Triangle 中，在对 Shape 中定义的两个方法 draw()和 erase()实现时，要在声明中增加 public，否则这些类对接口方法的实现将缩小访问权限，会出编译时错误。

5.4.4　多重继承

在 C++中，多重继承要将多个父类合并到一个类中。因为每个父类都有自己的一套实现细节，导致合并操作复杂，并可能存在同一个方法的两种不同实现，由此产生代码冲突，增加代码的不可靠性。Java 中规定一个类只能继承一个父类，但可以实现多个接口，Java 是利用接口实现多重继承的。由于接口根本没有实现细节，所以在进行父类与多个接口的合并时，只可能有一个类具有实现细节，如图 5-3 所示。由此 C++多重继承实现中存在的问题，在 Java 中都不存在了，保证了 Java 的简单性与代码的安全可靠。

下面举一个多重继承的例子。

对于 5.4.3 节中给出的描述飞行行为的接口 Flyer，由于飞机、鸟，甚至科幻中的超人都可以飞，所以 Airplane 类、Bird 类和 Superman 类都可以实现 Flyer 接口。而同时，飞机是一种交通工具，因此 Airplane 类又是 Vehicle 类的子类；鸟是一种动物，因此 Bird 又是 Animal 类的子类，如图 5-4 所示。所以一个类可以从一个父类继承，并且可以同时继承其他接口。

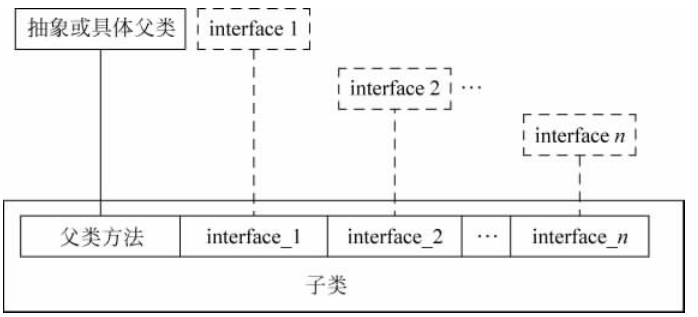

图 5-3 Java 中的多重继承

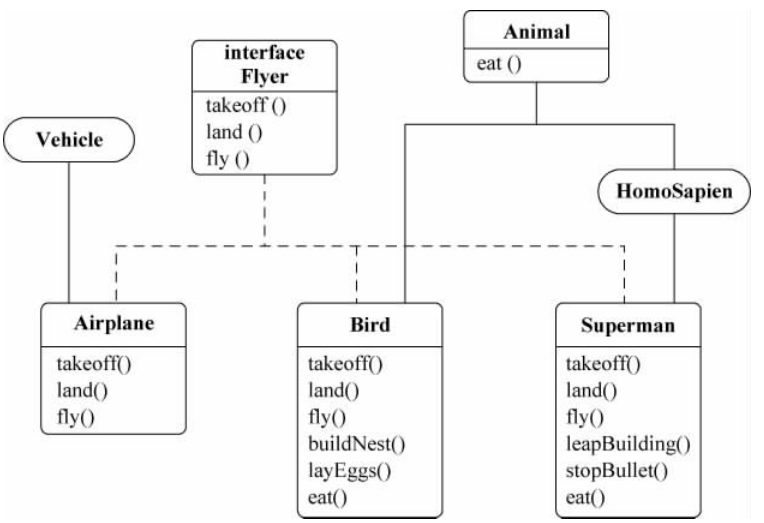

图 5-4 类体系中的类同时实现接口的示例

Bird 类可以进行如下定义:

```
public class Bird extends Animal implements Flyer{
  public void takeoff(){…}
  public void land(){…}
  public void fly(){…}
  public void buildNest(){…}
  public void layEggs(){…}
  public void eat(){…}
}
```

注意:在子类的声明中,extends 子句必须放在 implements 子句前面。

一个类也可以实现多个接口。水上飞机(Seaplane)不仅能飞还能够在海上航行。Seaplane 类继承了 Airplane 类,所以继承了 Airplane 类中对 Flyer 接口的实现,从而具有了飞行的行为;而同时 Seaplane 类也可以实现 Sailer 接口,使该类具有航行的接口与行为,如图 5-5 所示。

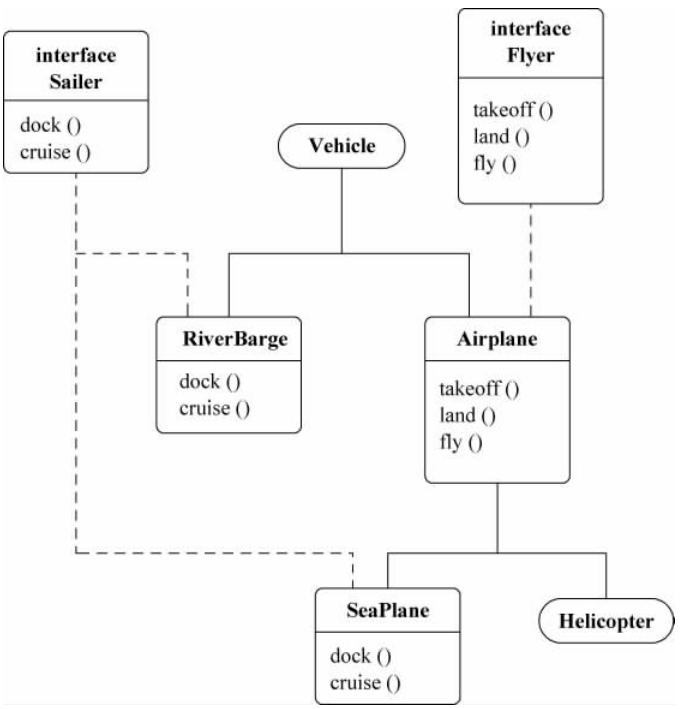

图 5-5　类体系中的类同时实现多个接口

5.4.5　通过继承扩展接口

接口定义后，可能在某些情况下需要对接口进行扩展，如增加新的方法声明。例如，对于例 5-5 中的接口 Shape，如果需要计算一个几何图形的面积，可以向 Shape 中加入一个方法：

```
interface Shape {
    void draw();
    void erase();
    double area();
}
```

上述直接向 Shape 中增加方法的方式扩展接口可能带来问题：所有实现原来 Shape 接口的类都将因为 Shape 接口的改变而不能正常工作。为了既能扩展接口，又保证不影响实现该接口的类，一种可行的方法是通过创建接口的子接口来增加新的方法，例如：

```
interface ShapeArea extends Shape{
      double area();
}
```

这样使用 Shape 接口的用户可以选择采用新的接口 ShapeArea，也可以保持原来对 Shape 接口的实现。

在接口的定义中使用继承，可以方便地为一个接口添加新的方法；也可以通过接口继承将几个接口合并为一个接口，即在子接口声明中的 extends 关键字后引用多个基础接口，

这些接口间通过“,”分隔。

5.4.6 接口与抽象类

通过上述对抽象类和接口的介绍,可以发现接口与抽象类有一定的相似性,但实际上这二者之间有很大的区别,如下所述。

(1) 接口中的所有方法都是抽象的,而抽象类可以定义带有方法体的不同方法。

(2) 一个类可以实现多个接口,但只能继承一个抽象父类。

(3) 接口与实现它的类不构成类的继承体系,即接口不是类体系的一部分。因此,不相关的类也可以实现相同的接口。而抽象类是属于一个类的继承体系,并且一般位于类体系的顶层。

使用接口的主要优势在于:一是类通过实现多个接口可以实现多重继承,这是接口最重要的作用,也是使用接口的最重要的原因——能够使子类对象上溯造型为多个基础类(接口)类型。另一个优势是能够抽象出不相关类之间的相似性,而没有强行形成类的继承关系。使用接口,可以同时获得抽象类以及接口的优势。所以如果要创建的类体系的基础类不需要定义任何成员变量,并且不需要给出任何方法的完整定义,则应该将基础类定义为接口。只有在必须使用方法定义或成员变量时,才应该考虑采用抽象类。

5.5 包

5.5.1 什么是 Java 中的包

在 Java 中,为了使类易于查找和使用,为了避免命名冲突和限定类的访问权限,可以将一组相关类与接口“包裹”在一起形成包(package)。包被认为是 Java 的重要特色之一,它体现了 OOP 的封装思想,为 Java 中管理大量的类和接口提供了方便。另外,由于 Java 编译器要为每个类生成一个字节码文件,且文件名与类名相同,因此有可能由于同名类的存在而导致命名冲突。包的引入为 Java 提供了以包为单位的独立命名空间,位于不同包中的类即使同名也不会冲突,从而有效地解决了命名冲突的问题。同时,包具有特定的访问控制权限,同一个包中的类之间拥有特定的访问权限。因此,Java 中包是相关类与接口的一个集合,它提供了类的命名空间的管理和访问保护。

Java 平台中的类与接口都是根据功能以包组织的。Java 的 JDK 提供的包主要有:

java.applet,java.awt,java.awt.datatransfer,java.awt.event,java.awt.image,java.beans,java.io,java.lang,java.lang.reflect,java.math,java.net,java.rmi,java.security,java.sql,java.util 等。

每个包中都定义了许多功能相关的类和接口。我们也可以定义自己的包来实现自己的应用程序。

Java 编译器把包对应于文件系统的目录和文件管理,还可以使用 ZIP 或 JAR 压缩文件的形式保存。例如,以 Windows 平台为例,名为 java.applet 的包中,所有类文件都存储在目录 classPath\java\applet 下。其中包根目录——classPath 由环境变量 CLASSPATH 来设定。

包机制的好处主要体现在如下几点。

- 程序员容易确定包中的类是相关的，并且容易根据所需的功能找到相应的类。
- 每个包都创建一个新的命名空间，因此不同包中的类名不会冲突。
- 同一个包中的类之间有比较宽松的访问控制。

下面将介绍如何定义与使用包。

5.5.2　包的定义与使用

1. 包的定义

使用 package 语句指定一个源文件中的类属于一个特定的包。package 语句的格式如下：

```
package pkg1[.pkg2[.pkg3…]];
```

例如：

```
package graphics;
public class Circle extends Graphic implements Draggable {
    …
}
```

Circle 类成为 graphics 包中的一个 public 成员，并存放在 classPath\graphics 目录中。如果源文件中没有 package 语句，则指定为无名包。无名包没有路径，一般情况下，会把源文件中的类存储在当前目录（即存放 Java 源文件的目录）下。前面许多例子都属于这种情况。

说明：

（1）package 语句在每个 Java 源程序中只能有一条，一个类只能属于一个包。

（2）package 语句必须在程序的第一行，该行前可有空格及注释行。

（3）包名以“.”为分隔符。

2. 包成员的使用

包中的成员是指包中的类和接口。只有 public 类型的成员才能被包外的类访问。要从包外访问 public 类型的成员，要通过以下方法。

- 引入包成员或整个包，然后使用短名（short name，类名或接口名）引用包成员。
- 使用长名（long name，由包名与类/接口名组成）引用包成员。

（1）引入包成员

可以先引入包中的指定类或整个包，再使用该类。这时可以直接使用类名或接口名。在 Java 中引入包（如 JDK 中的包或用户自定义的包）中的类是通过 import 语句实现的。import 语句的格式如下：

```
import pkg1[.pkg2[.pkg3…]].(classname|*);
```

其中 pkg1[.pkg2[.pkg3…]]表明包的层次，与 package 语句相同，它对应于文件目录，classname 则指明所要引入的类。如果要从一个包中引入多个类，则可以用通配符（*）来代替。例如下列代码引入 graphics 包中的指定类 Circle：

```
import graphics.Circle;      //引入 graphics 包中的 Circle 类
```

```
…
Circle myCircle = new Circle();
…
```

下列代码引入 graphics 包中的所有类,程序中便可以直接引用该包中的任意类,如 Circle 和 Rectangle:

```
import graphics.*;        //引入 graphics 包中的所有类
…
Circle myCircle = new Circle();
Rectangle myRectangle = new Rectangle();…
```

注意:import 语句必须在源程序所有类声明之前,在 package 语句之后。因此 Java 程序的一般结构如下:

```
[ package 语句]        //默认是 package.;(属于当前目录)
[ import 语句]         //默认是 import  java.lang.*;
[类声明]
```

(2) 使用长名引用包成员

要在程序中使用其他包中的类,而该包并没有引入,则必须使用长名引用该类。长名的格式是:

```
包名.类名
```

例如,如果当前程序要访问 graphics 包中的 Circle 类,但该类并未通过 import 语句引入,则要使用 graphics.Circle 来引用 Circle 类:

```
…
graphics.Circle  myCircle = new graphics.Circle();
…
```

这种方式过于烦琐,一般只有当两个包中含有同名的类时,为了对两个同名类加以区分才使用长名。如果没有这种需要,更简单常用的方法是使用 import 语句来引入所需要的类,然后在随后的程序中直接使用类名对类操作。

3. 包定义与使用示例

例 5-6 定义二维几何图形的包并使用。

(1) 文件 Rectangle.java。定义了 Rectangle 类放入 graphics.twoD 包中。

```
package graphics.twoD;
public class Rectangle {
    public int width = 0;
    public int height = 0;
    public Point origin;

    public Rectangle(Point p, int w, int h) {
        origin = p;
        width = w;
        height = h;
    }
```

```
    //移动矩形的方法
    public void move(int x, int y) {
        origin.x = x;
        origin.y = y;
    }

    //计算矩形面积的方法
    public int area() {
        return width * height;
    }
}
```

(2) 文件 Point.java。定义了 Point 类放入 graphics.twoD 包中。

```
package graphics.twoD;
public class Point {
    public int x = 0;
    public int y = 0;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

(3) 文件 TestPackage.java,包含 main()方法的测试程序。定义了一个点及一个矩形,计算并输出矩形的面积。

```
import graphics.twoD.*;
public class TestPackage{
    public static void main(String args[]){
        Point p = new Point(2,3);
        Rectangle r = new Rectangle(p,10,10);
        System.out.println("The area of the rectangle is " + r.area());
        }
    }
```

假如例 5-6 中 Point.java 与 Rectangle.java 放在 c:\work 下,TestPackage.java 在 c:\work\test 目录下,而 graphics.twoD 将位于 c:\mypkg 下,则例 5-6 的编译与运行可按下列步骤进行。

(1) 将 c:\mypkg 添加到 classpath 系统变量中,使该路径作为一个包根路径。既可以通过 set 命令添加,即 set classpath = %classpath%; c:\mypkg,也可以在 Windows 中通过系统变量的设置窗口进行。

(2) 将 c:\work 作为当前目录,输入:

```
javac -d c:\mypkg Point.java Rectangle.java
```

则在 c:\mypkg\graphics\twoD 目录下将产生 Point.class 和 Rectangle.class 两个类文件。Javac 命令中的-d 选项是指定编译所产生类文件的根路径,如不指定,则编译生成的类文件如 Point.class 和 Rectangle.class 将存放在当前路径下。

(3) 进入 TestPackage.java 所在的目录 c:\work\test,先后输入下列命令编译和运行:

```
javac TestPackage.java
java TestPackage
```

TestPackage.java 的运行结果如下:

```
The area of the rectangle is 100
```

5.5.3 引入其他类的静态成员

如果程序中需要频繁使用其他类中定义为 static final 的常量或 static 方法,则每次引用这些常量或方法都要出现它们所属的类名,会使得程序显得比较混乱。Java 中提供了 static import 语句,使得程序中可以引入所需的常量和静态方法,这样程序中对它们的使用就不用再带有类名了。

例如,java.lang.Math 类定义了一个常量 PI 和很多静态方法,包括计算正弦、余弦、正切、余切等的方法:

```
public static final double PI 3.141592653589793
public static double cos(double a)
```

在 JDK 1.5 之前的版本中,如果在其他程序中要使用它们,则需要在这些常量和方法前带有所属类名:

```
double r = Math.cos(Math.PI * theta);
```

现在,程序可以使用 static import 语句,将 java.lang.Math 类的 static 成员引入,则在程序中使用 Math 的静态成员将不再需要带有类名。静态成员可以单个引入,也可以通过通配符“*”成组引入,例如:

```
import static java.lang.Math.PI;
```

或

```
import static java.lang.Math.*;
```

一个类的静态成员一旦被引入后,就可以直接使用成员的名称,而不用带类名。例如上面对 Math 类的 cos()方法的调用,可以用下列代码替代:

```
double r = cos(PI * theta);
```

对于 JDK 类库之外的用户自定义类的静态成员,也可以使用这种静态引入语句。

上述静态引入语句如果使用得当,会使程序变得简明易读,但如果过多使用则会适得其反。除了程序员,其他人很难了解哪些类定义了哪些静态成员,所以,程序中过多使用静态引入会使程序变得难以理解和维护。

5.5.4 包名与包成员的存储位置

从例 5-6 可以看到,Java 中包名实际上是包的存储路径的一部分,包名中的.分隔符相当于目录分隔符。包存储的路径实际上由包根路径加上包名指明的路径组成,而包的根路

径由 CLASSPATH 环境变量指出。

假如 CLASSPATH 环境变量按下面的值进行设置：

```
CLASSPATH = c:\jdk1.4.2\lib; .; c:\mypkg
```

则 Java 在编译 TestPackage.java 和解释执行 TestPackage.class 时，将会在下列路径下查找 Point 类和 Rectangle 类，下面以 Point.class 为例。

(1) c:\jdk1.4.2\lib\graphics\twoD\Point.class

(2) .\graphics\twoD\Point.class

(3) c:\mypkg\graphics\twoD\Point.class

如果在上述路径下都没有找到 Point 类，则将产生编译或运行时错误。

5.5.5　Java 源文件与类文件的管理

利用 Java 的包名与类文件存储位置之间的关系，可以对 Java 应用程序的源文件与类文件进行很好的管理。下面几点是进行一个应用系统开发时可以参考的。

(1) 在应用系统目录下分别创建源文件目录与类文件目录，并把类文件目录加入到 classpath 环境变量中。

例如，要开发几何图形操作相关的应用，可以创建下列目录：

d:\graphicApp\source——作为存放源文件的顶层(根)路径。

d:\graphicApp\classes——作为存放类和接口的文件的包根路径。

(2) 每个源文件都存放在 source 目录中以包名为相对路径的子目录下；编译后产生的类文件以所属包名为相对路径，存储在 classes 目录下。例如对于例 5-6 的程序，其源文件与类文件可以保存在如图 5-6 的层次目录下。

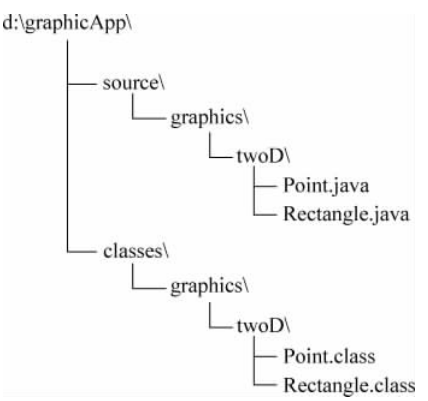

图 5-6　graphicApp 的源文件与类文件的管理

由于 Java 程序在编译时，每个类都要生成一个文件，所以一个应用程序包含的文件可能是很多的。按照上述方法对 Java 应用系统的文件进行存放，可以实现对文件的有效管理，并且能够保证引用这些类的程序在编译和运行时能够简便有效地定位到相应的类。

5.6　泛型与集合类

5.6.1　泛型概述

泛型即泛化技术(generics)，是在 JDK 1.5 中引入的重要语言特征。泛型技术可以通过一种类型或方法操纵各种类型的对象，而同时又提供了编译时的类型安全保证。在 JDK 1.5 以后的版本中，Java 对 JDK 中的集合类(collections)应用了泛型。

在软件开发中，程序中的错误几乎是难以避免的。这些错误可以分为编译时错误与运

行时错误。编译时错误发现早,可以根据编译器提示的错误信息比较容易进行修改,而运行时错误却难以定位和修改。泛型使得很多程序中的错误能够在编译时刻被发现,从而增加了代码的正确性和稳定性。

泛型技术的基本思想是类和方法的泛化,是通过参数化实现的,因此泛型又被称为参数化类型,即通过定义含有一个或多个类型参数的类或接口,使得程序员可以对具有类似特征与行为的类进行抽象。下面通过一个例子说明泛型的概念。

在 JDK 1.4 及以前版本中,因为集合类可以保存 Object 及其子类的对象,所以可以创建一个集合类的实例,如 LinkedList,存放各种类型的对象。程序员对 LinkedList 中的对象的具体类型是清楚的,并且要通过强制类型转换才能使用某些特定于对象的操作。有时,程序员也可以通过注释或其他说明性手段说明 LinkedList 中保存的对象类型。但是,编译程序是无法通过这些程序或注释了解到集合中对象类型的任何信息,也就无法进行类型检查,因此容易发生运行时错误。如例 5-7 所示。

例 5-7 不使用泛型的集合类示例。

```
1   import java.util.*;
2   public class ListTest {
3       public static void main(String[] args) {
4
5           //注意: 列表中只存放 Integer 类型的变量
6           List listofInteger = new LinkedList();
7           listofInteger.add(new Integer(2000));
8           listofInteger.add("8");
9
10          Integer x = (Integer) listofInteger.get(0);
11          System.out.println(x);
12          x = (Integer) listofInteger.get(1);
13          System.out.println(x);
14      }
15  }
```

例 5-7 的第 8 行中,程序员误将数字 8 以字符串的形式放到了 listofInteger 中,而在第 12 行,将“8”取出后,执行 Integer 的强制类型转换,这是错误的。例 5-7 会通过编译,但在运行时将会出现错误信息,如图 5-7 所示。

图 5-7 例 5-7 的运行结果

如果使用泛型编写上述程序,将程序员对 LinkedList 中对象类型的意图传递给编译器,则例 5-7 中程序员的疏忽会在编译时就被检查出。例 5-8 是采用泛型编写的程序。

例 5-8 使用泛型的集合类使用测试。

```
1   import java.util.*;
2   public class ListTestWithGenerics {
```

```
3       public static void main(String[] args) {
4
5         List<Integer> listofInteger = new LinkedList<Integer>();
6         listofInteger.add(new Integer(2000));
7         listofInteger.add("8");
8
9         Integer x = listofInteger.get(0);
10        System.out.println(x);
11        x = listofInteger.get(1);
12        System.out.println(x);
13      }
14  }
```

例 5-8 中，第 5 行变量 listofInteger 的声明 List＜Integer＞，表示它不是一个存放 Object 类型对象的列表，而是存放 Integer 对象的列表。此处，List 就是一个带有类型参数(Integer)的泛化接口，并且在创建这个列表对象时，也要指定类型参数。另外，例 5-8 中的第 9 行和第 11 行原有的强制类型转换也去掉了。

通过例 5-8 第 5 行中的声明，编译器了解了程序员对变量 listofInteger 类型限定的意图，并且将在编译时对程序进行相应的类型检查，并且保证所有对 listofInteger 变量操作满足其类型的要求。而强制类型转换只是表明程序员对某行代码的操作认为是正确的，并且无法实现编译时的检查。在例 5-8 的第 7 行中向 listofInteger 增加非 Integer 类型的对象，则在编译时会出现下列错误：

```
ListTestWithGenerics.java:7: 找不到符号
符号: 方法 add(java.lang.String)
位置: 接口 java.util.List＜java.lang.Integer＞
        listofInteger.add("8");
                      ^
1 错误
```

将第 7 行代码改为 listofInteger.add(new Integer(8))或 listofInteger.add(8)，则例 5-8 可以通过编译并正常运行，结果如下：

```
2000
8
```

因此，泛型增加了程序的可读性和强壮性。

5.6.2　泛化类型及其子类

1. 泛化类型(泛型)的定义

进行泛型编程的基础是要定义泛化类型(generic type)即泛型，也就是定义具有泛化结构的类或接口。JDK 1.5 和 JDK 1.6 中的集合类(collection)都已经被定义为泛型，所以例 5-8 中才可以使用这些泛型。

泛型的定义与普通类定义相比，首先在类名后增加了由尖括号标识的类型变量，一般用 T 表示。T 可以在泛型中的任何地方使用。对于泛化接口也是这样定义。下列代码定义了普通类 Box 以及 Box 的泛型。

(1) 普通类 Box 的定义

```
public class MyBox {
        private Object object;
        public void add(Object object) {
            this.object = object;
        }
        public Object get() {
            return object;
        }
    }
```

(2) Box 类泛型的定义

```
public class MyBox<T> {
    private T t;
    public void add(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
}
```

在 MyBox 类的泛型定义中,将类声明中的"public class MyBox"改为"public class MyBox <T>",并且把 MyBox 类体中出现的所有 Object 都用 T 进行替换,从而将 MyBox 定义为能够存放各种确定类型对象容器的抽象类型。例如,当我们在代码中通过 MyBox<Integer>创建一个 MyBox 对象,则该对象就只能存放 Integer 类型的对象。从上述例子中也可以看出,泛型的定义并不复杂。可以将 T 看做是一类特殊的变量,该变量的值在使用时指定,可以是除了基本数据类型之外的任意类型,包括类、接口,甚至可以是一个类型变量。T 可以被称为是一种类型形参,或类型参数。

泛型在使用时,必须像方法调用一样执行"泛型调用",将泛型中的类型变量 T 替换为具体的类、接口等,如例 5-9 所示。

例 5-9 泛型类的定义及其使用示例。

```
class MyBox<T> {
    private T t;
    public void add(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
}

public class MyBoxTest{
    public static void main(String args[]){
        MyBox<Integer> aBox;
        aBox = new MyBox<Integer>();
```

```
        aBox.add(new Integer(1000));
        Integer i = aBox.get();
        System.out.println("The Integer is : " + i);
    }
}
```

例 5-9 的运行结果是：

```
The Integer is : 1000
```

例 5-9 中出现对泛型 MyBox 的一个调用：

```
MyBox<Integer> aBox;
```

MyBox<Integer>读作“MyBox of Integer”。泛型调用与普通的方法调用相类似，所不同的是泛型调用时传递的实参是一个具体的类型而不是普通意义上的实参值。泛型的一个调用使泛型被固定为参数 T 所指定的类型，所以一般被称为参数化类型(parameterized type)。参数化类型的实例化还是使用 new 关键字，只是要在类名与“()”之间插入带有尖括号的参数类型。例如：

```
aBox = new MyBox<Integer>();
```

需要注意的是，泛型中的类型变量自身并不是实际存在的类型，即根本不存在 T.java 或 T.class，并且 T 也不是泛型类名的一部分。另外，一个泛型可以有多个类型参数，但是每个参数在该泛型中应是唯一的。例如不能出现 MyBox<T,T>，但可以出现 MyBox<T,U>。

2. 类型参数的命名习惯

习惯上，类型参数的名称用单个大写字母表示。这使得类型参数能够与其他变量名或类名、接口名有明显的区别。最常用的类型参数名包括如下几种。

- E——Element，表示元素，一般在 JDK 的集合类中使用。
- K——Key，表示键值。
- N——Number，表示数字。
- T——Type，表示类型。
- V——Value，表示值。
- S，U，V 等——可被用作一个泛化类型的第二个，第三个，第四个类型参数。

3. 泛型中的子类

在 Java 中，父类的变量可以指向子类的对象，因为子类被认为是与父类兼容的类型。因此，下列的代码是合法的：

```
Object someObject = new Object();
Integer someInteger = new Integer(10);
someObject = someInteger;
```

在泛型中，这一点仍然是成立的。可以使用一个父类作为类型参数调用泛型，而在后续对参数化类的访问中，使用该父类的子类对象。例如：

```
MyBox<Number> box = new MyBox<Number>();
box.add(new Integer(10));
```

```
box.add(new Double(10.1));
```

因为 Integer 和 Double 是 Number 的子类,所以上述代码是合法的。但在泛型中,MyBox<Number>与 MyBox<Integer>和 MyBox<Double>之间没有父子类关系,即 MyBox<Integer>和 MyBox<Double>不是 MyBox<Number>的子类。

首先从语义上,虽然 Integer 和 Double 是 Number 的子类,但容纳 Integer 和 Double 对象的 MyBox 却不一定是容纳 Number 对象 MyBox 的子类。再如,假设 Animal 是 Lion,Butterfly 的父类,而 Cage<Animal>,Cage<Lion>,Cage<Butterfly>分别是关养所有类型动物、狮子和蝴蝶的笼子。为了使各种类型的动物都不能逃跑,Cage<Animal>需要考虑各种情况,既要能够关住狮子、老虎,也要关住蝴蝶、蚯蚓,而 Cage<Lion>和 Cage<Butterfly>却只要能够关住狮子和蝴蝶就可以了,它们并没有 Cage<Animal>的所有特征并且要远比 Cage<Animal>简单。因此,Cage<Lion> 不是 Cage<Animal>的子类,而 Cage<Butterfly> 也不是 Cage<Animal>的子类。

另外,从 Java 泛型的语言机制方面,再进一步理解这个问题。假设 Java 中允许 List<String>是 List<Object>的子类,则程序中可以出现下列代码:

```
1  List<String>  ls = new ArrayList<String>();
2  List<Object> lo = ls;
3  Lo.add(new Object());
4  String s = ls.get(0);
```

但实际上上述代码是有问题的。第 4 行中,ls.get(0)返回的对象类型是 Object,而 String 类型的变量 s 是不能指向 Object 对象的,这是因为父类弱,子类强,父类中往往不包含子类的很多信息,所以不能按照子类的变量访问父类对象。根据第 1 行的定义,ls 指向一个字符串列表,如果没有第 2、3 行,第 4 行的代码是合法的。由于假设 List<String>是 List<Object>的子类,第 2 行代码合法并使得字符串列表有了另一个入口 lo。通过 lo 可以将 Object 类型的对象放入列表,从而出现通过 String 类型变量 ls 访问到 Object 对象的情况,所以 List<String>是 List<Object>子类的假设是不成立的。

因此,即使调用泛型的实参类型之间有父子类关系,调用后得到的参数化类型之间也不会具有同样的父子类关系。

5.6.3 通配符

Java 允许在泛型的类型形参中使用通配符(wildcards),以提高程序的灵活性。下面通过例子说明泛型中通配符的作用。

如果我们要编写一个方法实现给笼子里的动物喂食的操作,可以编写一个 feedAnimals()方法:

```
void feedAnimals(Cage<Animal> someCage) {
      for (Animal a : someCage)
      a.feedMe();
}
```

但上述 feedAnimals()方法,实际上只能对 Cage<Animal>中的动物喂食,给狮子和蝴蝶的喂食却不能调用该方法,因为 Cage<Lion> 和 Cage<Butterfly>并不是 Cage<Animal>的

子类。在这里需要定义所有动物笼子的父类。

Java 泛型中，提供了通配符实现这种类的定义：以通配符“?”替代泛型尖括号中的具体类型，表明该泛型的类型是一种未知的类。例如 Cage<?>，表示一种未知类型物体的笼子，可以指任何存放物体的笼子，可能是一种 Animal 的笼子，也可能是 Lion 的笼子，还可能是放某种水果 Fruit 的笼子。Cage<?>可以认为是 Cage<Animal>，Cage<Butterfly>，Cage<Fruit>的父类。

在上述 feedAnimals()中，需要定义所有动物笼子的父类。这可以通过使用受限通配符(Bounded wildcard)来实现：

```
Cage<? extends Animal>
```

“? extends Animal”的含义是 Animal 或其某种未知的子类，也可以理解为“某种动物”。Cage<? extends Animal>泛指 Animal 及其子类的笼子，是 Cage<Butterfly>和 Cage<Lion>的父类。在“? extends Animal”中，Animal 被认为是泛型变量的上限。也可以用 super 关键字代替 extends 定义泛型变量的下限：<? super Animal>，其含义是 Animal 或其未知的某个父类。在上文的 Cage<? >中，<? >称为无限制通配符(unbounded wildcards)，实际上是与<? extends Object>等价的。例 5-10 中给出了一个完整示例。

例 5-10　泛型中的通配符示例。

```
import java.util.*;
class Cage<E> extends LinkedList<E>{};
class Animal{
  public void feedMe(){ };
}
class Lion extends Animal{
    public void feedMe(){
        System.out.println("Feeding lions");
    }
}

class ButterFly extends Animal{
    public void feedMe(){
        System.out.println("Feeding butterflies");
    }
}

public class WildcardsTest{
    public static void main(String args[]){
        WildcardsTest t = new WildcardsTest();
        Cage<Lion> lionCage = new Cage<Lion>();
        Cage<Butterfly> butterflyCage = new Cage<Butterfly>();
        lionCage.add(new Lion());
        butterflyCage.add(new Butterfly());
        t.feedAnimals(lionCage);
        t.feedAnimals(butterflyCage);
    }
    void feedAnimals(Cage<? extends Animal> someCage) {
```

```
        for (Animal a:someCage)
            a.feedMe();
    }
}
```

例 5-10 的运行结果如下：

```
Feeding lions
Feeding butterflies
```

5.6.4 泛化方法

Java 泛型中，类型参数还可以出现在方法声明中，以定义泛化方法(generic methods)。泛化方法与泛型的声明类似，但泛化方法中类型参数的作用域只限于声明它的方法。下面例 5-11 中给出了一个泛化方法的例子。

例 5-11 泛化方法示例。

```
class MyBox<T> {
    private T t;
    public void add(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
    public <U> void inspect(U u){
        System.out.println("  T: " + t.getClass().getName());
        System.out.println("  U: " + u.getClass().getName());
        System.out.println();
    }
}

public class BoxTest{
    public static void main(String[] args) {
        MyBox<Integer> integerBox = new MyBox<Integer>();
        integerBox.add(new Integer(10));

        System.out.println("The first inspection:");
        integerBox.inspect("some text");

        System.out.println("The second inspection:");
        integerBox.inspect(new Double(100.0));
    }
}
```

例 5-11 的运行结果如下：

```
The first inspection:
  T: java.lang.Integer
  U: java.lang.String
```

```
The second inspection:
  T: java.lang.Integer
  U: java.lang.Double
```

例 5-11 中，inspect()定义了一个类型参数 U，是一个泛化方法。该方法把一个对象的类型打印到标准输出上。为了进行对比，该方法也输出了将类型变量 T 所指向对象的类型。

泛化方法的定义是在一般方法声明中增加了类型参数的声明。具体是在方法声明的各种修饰符，如 public，final，static，abstract，synchronized 等，与方法返回类型之间，增加一个带尖括号的类型参数列表。类型参数表的定义与泛型中的定义一样，也可以使用受限类型参数。

Java 中，不仅可以对实例方法进行泛化，也可以对静态方法、构造方法进行泛化，即所有方法都可以定义为泛化方法。

注意，在例 5-11 的 main()中，在对泛化方法 inspect()的调用时，并没有显式传递实参的类型，只是像普通的方法调用一样用实参去调用该方法。这主要是 Java 编译器具有类型推理的能力，它根据调用方法时实参的类型，推理得出被调用方法中类型变量的具体类型，并据此检查方法调用中类型的正确性。

泛化方法实现的功能，有时也可以用带有通配符的泛型实现。例如在下列代码中，containsAll()与 addAll()两个方法都定义为泛化方法：

```
interface Collection<E> {
    public <T> boolean containsAll(Collection<T> c);
    public <T extends E> boolean addAll(Collection<T> c);
}
```

但这两个方法也可以利用通配符定义：

```
interface Collection<E> {
    public boolean containsAll(Collection<?> c);
    public boolean addAll(Collection<? extends E> c);
}
```

那么泛化方法与通配符分别适合怎样的应用呢？从前面对通配符的介绍中可以看到，引入通配符的主要目的是支持泛型中的子类，从而实现多态。如果方法泛化的目的只是为了能够适用于多种不同类型，或支持多态，则应该使用通配符。泛化方法中类型参数的优势是可以表达多个参数或返回值之间的类型依赖关系。如果方法中并不存在类型之间的依赖关系，则可以不使用泛化方法，而使用通配符。在上述 Collection<E>的泛化方法定义中，类型参数 T 只使用了一次，并且 containsAll()与 addAll()只有一个参数，返回值也不依赖于类型参数，因此适宜采用通配符方式。

当然，泛化方法与通配符也可以一起使用，如下面的代码：

```
class Collections {
    public static <T> void copy(List<T> dest, List<? extends T> src){…}
}
```

一般地，因为通配符更清晰、简明，因此建议尽量采用。另外，通配符还可以在方法声明

之外使用,例如类变量、局部变量与数组的类型声明等。

5.6.5 类型擦除

Java 虚拟机中,并没有泛型类型的对象。泛型是通过编译器执行一个被称为类型擦除(type erases)的前端转换来实现的。类型擦除可以理解成一种源程序到源程序的转换,即把带有泛型的程序转换为不包含泛型的版本。

一般地,擦除进行下列处理。

(1) 用泛型的原生类型替代泛型。

原生类型(raw type)是泛型中去掉尖括号及其中的类型参数的类或接口。泛型中所有对类型变量的引用都替换为类型变量的最近上限类型,如对于 Cage<T extends Animal>,T 的引用将用 Animal 替换,而对于 Cage<T>,T 的引用将用 Object 替换。

例如,对于例 5-9 中定义的泛型 MyBox<T>,类型擦除后得到了相应的原生类型 MyBox:

```
public class MyBox {
    private Object t;
    public void add(Object t) {
        this.t = t;
    }
    public Object get() {
        return t;
    }
}
```

MyBox 是一个普通类,与泛型引入之前的类一样。

(2) 对于含泛型的表达式,用原生类型替换泛型。

例如,List<String>的原生类型是 List。类型擦除中,List<String>被转换成 List。对于泛型方法的调用,如果擦除后返回值的类型与泛型声明的类型不一致,则会插入相应的强制类型转换。

(3) 对于泛化方法的擦除,是将方法声明中的类型参数声明去掉,并进行类型变量的替换。

例如,对于方法:

```
public static <T extends  Comparable> T  min(T[] a)
```

类型擦除后,转换为:

```
public static  Comparable  min(Comparable [] a)
```

下面给出一个泛型的类型擦除示例。对于下列 loophole()方法:

```
public String loophole(Integer x) {
    List<String> ys = new LinkedList<String>();
    ys.add(x.toString());
    return ys.iterator().next();
}
```

经编译器进行泛型类型擦除后，实际运行的代码如下：

```
public String loophole(Integer x) {
    List ys = new LinkedList();
    ys.add(x);
    return (String) ys.iterator().next();
}
```

Java 虚拟机对于泛型采用擦除机制的主要目的，是为了与 JDK 1.5 之前的已有代码兼容。在 JDK 1.5 与 JDK 1.6 中，对于 JDK 中定义的泛型如集合类，应尽量使用泛型机制，而不要使用原生类。如果将泛型与原生类混合使用，编译器会给出一些类型未检查的警告。对于这样的警告应给予重视并进行程序检查，否则就有可能出现运行时错误。

5.6.6　集合类

1. 集合类概述

一个集合对象或一个容器表示了一组对象，集合中的对象称为元素。在这个对象中，存放指向其他对象的引用。Java 的 Collections API 包括了下列核心集合接口，如图 5-8 所示。

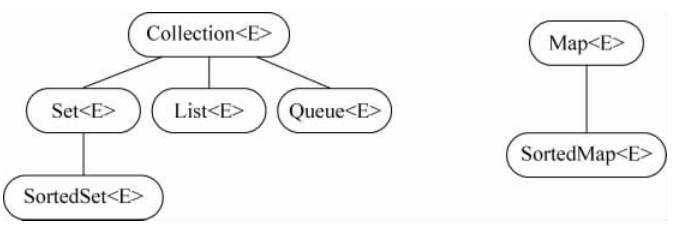

图 5-8　Java Collections API 的核心接口

图 5-8 中的 Java Collections API 的核心接口支持泛型，并且形成了两个独立的树形结构。Map 是一种特殊的集合，与一般的集合不同。

(1) Collection

Collection 接口是集合接口树的根，它定义了集合操作的通用 API。对 Collection 接口的某些实现类允许有重复元素，而另一些不允许有重复元素；某些是有序的而另一些是无序的。JDK 中没有提供这个接口的实现，而是提供了它的子接口如 Set 和 List 的实现。

(2) Set

Set 中不能包含重复的元素。它是数学中“集合”概念的抽象，可以用来表示类似于学生选修的课程集合或机器中运行的进程集合等。

(3) List

List 是一个有序的集合，称为列表或序列。List 中可以包含重复的元素。可以通过元素在 List 中的索引序号访问相应的元素。Vector 就是一种常用的 List。

(4) Map

Map 实现键值到值的映射。Map 中不能包含重复的键值，每个键值最多只能映射到一个值。Hashtable 就是一种常用的 Map。

(5) Queue

Queue 是存放等待处理的数据的集合，称为队列。Queue 中的元素一般采用 FIFO(先

进先出)的顺序,也有以元素的值进行排序的优先队列。无论队列采用什么样的顺序,remove()和 poll()方法都是对队列的最前面元素进行操作。在 FIFO 队列中,新添加的元素都是放到队列的尾部,其他队列可能采用不同的排放策略。

(6) SortedSet 和 SortedMap

SortedSet 和 SortedMap 分别是具有排序性能的 Set 和 Map。

2. 几种常用集合

(1) Set

Set 继承了 Collection 接口,Set 的方法都是从 Collection 继承的,它没有声明其他方法。Set 接口中所包含的方法如下,实现 Set 的类也实现了这些接口,所以我们可以对具体的 Set 对象调用这些方法:

```
public interface Set<E> extends Collection<E> {
    //基本操作
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);
    boolean remove(Object element);
    Iterator<E> iterator();              //返回当前集合元素的反复器 iterator

    //集合元素批操作
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);   //将集合 c 的元素都加到本集合中,成功返回
                                                 //true,否则为 false
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);          //在当前集合中只保留属于 c 的元素,如果当
                                                 //前集合发生变化则返回 true,
                                                 //否则返回 false
    void clear();                                //清除集合中的所有元素
    //数组操作
    Object[] toArray();      //返回包含当前集合所有元素的数组。该数组是新创建的,与当前集
                             //合独立。因此调用者可以随意修改返回的数组
    <T> T[] toArray(T[] a); //返回包含当前集合所有元素的数组。所返回数组的运行时类型是数
                             //组 a 的类型。如果数组 a 能够容下集合的所有元素,则将集合元素
                             //写入 a 并返回,否则创建类型与 a 相同、长度等于集合长度的数组
}
```

JDK 中提供了实现 Set 接口的 3 个实用的类:HashSet 类、TreeSet 类和 LinkedHashSet 类。

HashSet 类是采用 Hash 表实现了 Set 接口。一个 HashSet 对象中的元素存储在一个 Hash 表中,并且这些元素没有固定的顺序。由于采用 Hash 表,所以当集合中的元素数量较大时,其访问效率要比线性列表快。

TreeSet 类实现了 SortedSet 接口,是采用一种有序树的结构存储集合中的元素。TreeSet 对象中元素按照升序排序。

LinkedHashSet 类实现了 Set 接口,采用 Hash 表和链表相结合的结构存储集合中的元素。LinkedHashSet 对象的元素具有固定的顺序,它集中了 HashSet 与 TreeSet 的优点,既能保证集合中元素的顺序又能够具有较高的存取效率。

下面给出一个 Set 的使用实例。

例 5-12　Set 的使用示例。

```
import java.util.*;
public class FindDups {
    public static void main(String args[]) {
        //创建一个 HashSet 对象,默认的初始容量是 16
        Set<String> s = new HashSet<String>();

        //将命令行中的每个字符串加入到集合 s 中,其中重复的字符串将不能加入,并被打印输出
        for (String a : args){
            if (!s.add(a))
                System.out.println("Duplicate detected: " + a);
        }
    //输出集合 s 的元素个数以及集合中的所有元素
        System.out.println(s.size() + " distinct words detected: " + s);
    }
}
```

如果在 Windows 的命令窗口中输入下列命令：

```
java FindDups I come I see I go
```

则例 5-12 的运行结果如下：

```
Duplicate detected: I
Duplicate detected: I
4 distinct words detected: [see, come, I, go]
```

(2) List

List 是一种有序的集合,它继承了 Collection 接口。除了继承了 Collection 中声明的方法,List 接口中还增加了如下操作。

- 按位置存取元素：按照元素在 List 中的序号对其进行操作。
- 查找：在 List 中搜寻指定的对象并返回该对象的序号。
- 遍历：使用了 ListIterator 实现对一个 List 的遍历。
- 子 List 的截取,即建立 List 的视图(view)：能够返回当前 List 中的任意连续的一部分,形成子 List。

List 接口的定义如下：

```
public interface List<E> extends Collection<E> {
    //按位置存取元素
    E get(int index);
    E set(int index, E element);
    Boolean add(E element);
    void add(int index, E element);
    E remove(int index);
    boolean addAll(int index, Collection<? extends E> c);
```

```
    //查找
    int indexOf(Object o);
    int lastIndexOf(Object o);

    //遍历
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    //子 List 的截取
    List<E> subList(int from, int to);
}
```

JDK 中提供了实现 List 接口的 3 个实用类：ArrayList 类、LinkedList 类和 Vector 类，这些类都在 java.util 包中。

ArrayList 类采用可变大小的数组实现了 List 接口。除了实现 List 接口，该类还提供了访问数组大小的方法。ArrayList 对象会随着元素的增加其容积自动扩大。这个类是非同步的(unsynchronized)，即如果有多个线程对一个 ArrayList 对象并发访问，为了保证 ArrayList 数据的一致性，必须在访问该 ArrayList 的程序中通过 synchronized 关键字进行同步控制。ArrayList 是 3 种 List 中效率最高也是最常用的。它还可以使用 System.Arraycopy()进行多个元素的一次复制。除了非同步特性之外，ArrayList 几乎与 Vector 类是等同的，可以把 ArrayList 看做是没有同步开销的 Vector。

LinkedList 类采用链表结构实现 List 接口。除了实现 List 接口中的方法，该类还提供了在 List 的开头和结尾进行 get，remove 和 insert 等操作。这些操作使得 LinkedList 可以用来实现堆栈、队列或双端队列。LinkedList 类也是非同步的。

Vector 类采用可变体积的数组实现 List 接口。该类像数组一样，可以通过索引序号对所包含的元素进行访问。它的操作方法几乎与 ArrayList 相同，只是它是同步的。

例 5-13 是一个使用 List 的例子。这是一个关于扑克牌的例子。该例中用 ArrayList 保存了 52 张扑克牌，并通过 Collections 类的 static 方法 shuffle()实现“洗牌”操作。最后利用 dealHand()方法为参加游戏的人每人生成一手牌，每手牌的牌数是指定的。该程序有两个命令行参数：参加纸牌游戏的人数以及每手牌的牌数。

例 5-13 List 的使用示例。

```
import java.util.*;
class Deal {
    public static void main(String args[]) {
        int numHands = Integer.parseInt(args[0]);
        int cardsPerHand = Integer.parseInt(args[1]);

        //生成一副牌(含 52 张牌)
        String[] suit = new String[] {"spades", "hearts", "diamonds", "clubs"};
        String[] rank = new String[]
            {"ace","2","3","4","5","6","7","8","9","10","jack","queen","king"};
        List<String> deck = new ArrayList<String>();
        for (String ss : suit)
            for (String sr : rank)
                deck.add(sr + " of " + ss);
```

```
        Collections.shuffle(deck);                    //随机改变 deck 中元素的排列次序,即洗牌
        for (int i = 0; i < numHands; i++ )
            System.out.println(dealHand(deck, cardsPerHand));//生成一手牌并将其输出
    }

    public static List dealHand(List<String> deck, int n) {
        int deckSize = deck.size();
        //从 deck 中截取一个子链表
        List<String> handView = deck.subList(deckSize - n, deckSize);
        List<String> hand = new ArrayList<String>(handView);//利用该子链表创建一个链表
        handView.clear();//将子链表清空
        return hand;
    }
}
```

如果在 Windows 命令窗口中输入下列命令：

```
java Deal 2 5
```

则例 5-13 的某次运行结果如下：

```
[4 of clubs, 4 of hearts, 8 of clubs, queen of clubs, 2 of hearts]
[7 of spades, 6 of hearts, 10 of diamonds, 6 of spades, 10 of spades]
```

例 5-13 每次运行的结果都可能是不一样的。

注意，List.subList()方法返回的子 List 称为当前 List 的视图(view)，这意味着对子 List 的改变将反映到原来的 List 中。所以例 5-13 的 dealHand()方法中，执行 handView.clear()方法将 handView 清空，同时也将 deck 中对应于 handView 的元素删除了。因此每次调用 dealHand()方法都将返回包含 deck 中后面指定数目的元素并把它们从 deck 中清除掉。

例 5-13 中用到的 java.util.Collections 类是一个集合操作的实用类。该类提供了集合操作的很多方法，如同步、排序、逆序等，而且所有方法都是 static，因此可以不通过实例化直接调用。例如，常用集合 Set，List 和 Map 的 put()，get()，remove()等方法是不同步的，如果有多个线程同时对一个集合对象进行操作，就可能导致集合对象数据的错误。所以必须对共享集合的操作实现同步控制，使得一个时间内只能有一个线程对集合进行操作，保证数据的一致性。Collection 类提供了集合对象同步控制，它提供了一系列方法使集合对象具有同步控制能力，例如调用 synchronizedList(List＜T＞ list)方法，将得到一个基于指定 list 的具有同步控制的 list。

(3) Queue

Queue 除了基本的 Collection 接口中定义的操作，还提供了其他插入、删除和元素检查等操作。队列可以限定其元素的个数，这样的队列称为有界队列。在 java.util.concurrent 包中的某些队列的实现是有界的，而在 java.util 包中队列的实现类是没有元素个数限制的。Queue 接口的定义如下：

```
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E e);
```

```
    E peek();
    E poll();
    E remove();
}
```

Queue 提供的插入、删除和元素检查等方法都有两种形式,每种形式执行的操作是一样的,只是在操作不能正常进行时的处理不一样:一种方式是抛出异常,另一种方式是返回 null 或 false 等特定值。关于这两种方式的具体说明,如表 5-1 所示。

表 5-1 队列操作的两种方式比较

队列操作	功能说明	异常情况	抛出异常的方法	返回特定值的方法
插入	向队列中加入元素	有界队列满	add(e)	offer(e),返回 false
移除	从队首移走一个元素	队列空	remove()	poll(),返回 null
元素检查	返回队首元素,但不删除该元素	队列空	element()	peek(),返回 null

例 5-14 是一个使用队列保存待处理数据的例子。程序实现了一个倒计数的计数器。具体处理流程是:先把从时间 time 到 0 的所有整数,按从大到小的顺序存储在队列 queue 中,然后每隔 1 秒钟从队列中移出一个数打印输出。

例 5-14 Queue 使用示例。

```
import java.util.*;
public class Counter {
    public static void main(String[] args){
        int time = 5;                                          //设定计时开始时间
        Queue<Integer> queue = new LinkedList<Integer>();      //创建队列
        for (int i = time; i >= 0; i--)
            queue.add(i);                                      //把整数秒数存储在队列中
        while (!queue.isEmpty()) {
            System.out.println("    " + queue.remove());
            try{
                Thread.sleep(1000);
            }catch(InterruptedException e){ }                  //把队列中的整数输出
        }
    }
}
```

例 5-14 的运行结果如下:

```
5
4
3
2
1
0
```

(4) Map

Map 包含了一系列“键(key)-值(value)”之间的映射关系。一个 Map 对象可以看成是一个“键-值”对的集合,可以在该集合中通过一个键找到其对应的值。“键”和“值”可以是任意类型的对象。

如图 5-8 所示，Map 接口是独立于 Collection 接口体系的，Map 体系中的所有类和接口的方法都源自 Map 接口。Map 接口的定义如下所示：

```
public interface Map<K,V> {
    //基本操作
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    //整体批操作
    void putAll(Map <? extends K, ? extends V> m);
    void clear();

    //集合视图
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();

    //为 entrySet 元素定义的接口
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

Map 接口的方法主要实现下列 3 类操作。

- 基本操作：包括向 Map 中添加值对，通过键获取对应的值或删除该“键-值”对，测试 Map 中是否含有某个键或某个值，以及返回 Map 包含元素个数等。
- 批操作：包括向当前 Map 中添加另一个 Map 和清空当前 Map 的操作。
- 集合视图：包括获取当前 Map 中键的集合、值的集合以及所包含的“键-值”对等。其中“键-值”对集合的元素类型由 Map 中的内部接口 Entry 定义。

JDK 中提供了实现 Map 接口的实用类，包括 HashMap 类、HashTable 类、TreeMap 类、WeekHashMap 类和 IdentityHashMap 类等。

HashMap 类和 HashTable 类都采用 Hash 表实现 Map 接口。HashMap 是无序的，它与 HashTable 几乎是等价的，区别在于 HashMap 是非同步的并且允许有空的键与值。由于采用 Hash 函数，对于 Map 的普通操作性能是稳定的，但如果使用 iterator 访问 Map，为了获得高的运行效率最好在创建 HashMap 时不要将它的容量设得太大。

TreeMap 类与 TreeSet 类相似，是采用一种有序树的结构实现了 Map 的子接口 SortedMap。该类将按键的升序的次序排列元素。

WeekHashMap 类与 HashMap 相类似，只是 WeekHashMap 中的“键-值”对在其键不再被使用时将自动被删除，由垃圾搜集器回收。

IdentityHashMap 类与其他 Map 类相比,其特殊之处是在比较两个键是否相同时,比较的是键的引用而不是键对象自身。

上述 Map 类中,HashMap(无序的 Map)和 TreeMap(有序的 Map)是常用的。

下面给出一个 Map 的使用实例。例 5-15 中,利用 TreeMap 进行单词词频的统计。将单词与该单词的词频作为"键-值"的映射对。

例 5-15 利用 Map 进行单词词频的统计。

```
import java.util.*;
public class Freq {
    public static void main(String args[]) {
        String[] words = {"if","it","is", "to", "be", "it", "is", "up",
                          "to", "me", "to", "delegate"};
        Integer freq;
        Map<String, Integer> m = new TreeMap<String, Integer>();

        //构造字符串数组 words 的单词频率表。以单词为 key,以词频为 value
        for (String a : words) {
          freq = m.get(a);              //获取指定单词的词频

            //词频递增
          if (freq==null){
            freq = new Integer(1);
          }else{
            freq = new Integer(freq.intValue() + 1);
          }
          m.put(a, freq);               //在 Map 中更改词频
        }
        System.out.println(m.size() + " distinct words detected:");
        System.out.println(m);
    }
}
```

例 5-15 的运行结果如下:

```
    8 distinct words detected:
{be=1, delegate=1, if=1, is=2, it=2, me=1, to=3, up=1}
```

3. 集合元素的遍历

Java Collections API 为集合对象提供了 iterator(重复器),用来遍历集合中的元素。Iterator 接口中的方法使我们可以向前遍历所有类型的集合。在对一个 Set 对象的遍历中,元素的遍历次序是不确定的。List 对象的遍历次序是从前向后,并且 List 对象还支持 Iterator 的子接口 ListIterator,该接口支持 List 的从后向前的反向遍历。

Iterator 层次体系中包含两个接口:Iterator 以及 ListIterator。它们的定义如下:

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
```

```
    void remove();
}

public interface ListIterator<E> extends Iterator<E> {
    boolean hasNext();
    E next();

    boolean hasPrevious();
    E previous();

    int nextIndex();
    int previousIndex();

    void remove();
    void set(E e);
    void add(E e);
}
```

Iterator 中的 remove()方法将删除当前遍历到的元素，即删除由最近一次 next()或 previous()调用返回的元素。

ListIterator 中的 set()方法可以改变当前遍历到的元素。add()方法将在下一个将要取得的元素之前插入新的元素。如果实际操作的集合不支持 remove()，set()或 add()方法，则将抛出 UnsupportedOperationException。

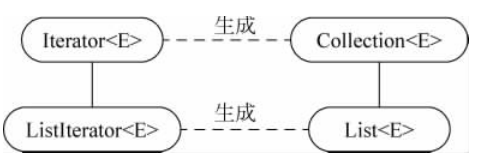

图 5-9　Iterator 层次结构图

图 5-9 表示了 Iterator 和 ListIterator 的继承关系，以及它们与 Collection 和 List 的关系。

例 5-16 是利用 ListIterator 操作一个 ArrayList 的例子。

例 5-16　ListIterator 的使用示例。

```
import java.util.*;
public class ListIteratorDemo {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();

        //向 list 中添加元素
        for(int i=1; i<5; i++){
            list.add(new Integer(i));
        }
        System.out.println("The original list : " + list);

        //创建 list 的 iterator
        ListIterator<Integer> listIter = list.listIterator();
        listIter.add(new Integer(0));              //在序号为 0 的元素前添加一个元素
        System.out.println("After add at beginning:" + list);

        if (listIter.hasNext()) {
```

```
            int i = listIter.nextIndex();          //i 的值将为 1
            listIter.next();                       //返回序号为 1 的元素
            listIter.set(new Integer(9));          //修改 list 中的序号为 1 的元素
            System.out.println("After set at " + i + ":" + list);
        }

        if (listIter.hasNext()) {
            int i = listIter.nextIndex();          //i 的值将为 2
            listIter.next();
            listIter.remove();                     //删除序号为 2 的元素
            System.out.println("After remove at " + i + " : " + list);
        }
    }
}
```

例 5-16 的运行结果如下：

```
The original list : [1, 2, 3, 4]
After add at beginning:[0, 1, 2, 3, 4]
After set at 1:[0, 9, 2, 3, 4]
After remove at 2 : [0, 9, 3, 4]
```

5.7 枚举类型

5.7.1 枚举概述

枚举类型(enum type)是在 JDK 1.5 以后引入的一种新的语法机制，一般用于表示一组常量。因此枚举定义中的域(field)是由固定的一组常量组成的，这些域的名称一般都使用大写字母。在 Java 中，应该尽量使用枚举类型表达固定不变的一组常量，例如方位(值为 NORTH，SOUTH，EAST 和 WEST)，一年中的季节(WINTER，SPRING，SUMMER，FALL)等，如例 5-17 所示。

例 5-17 枚举类型定义示例。

```
public enum Week {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
```

虽然最简单的 Java 枚举定义看起来与 C，C++和 C＃中的枚举定义很相像，但 Java 枚举类型的功能却比这些语言中的枚举强大得多。一个枚举的声明实际上是定义了一个类。这个类的声明中可以包含方法和其他属性，以支持对枚举值的相关操作，还可以实现任意的接口。枚举类型提供了所有 Object 类的方法，并且实现了 Comparable 和 Serializable 接口。

5.7.2 枚举类型的定义

枚举类型的定义格式与类的声明类似，一般格式如下：

```
[public] enum 枚举类型名 [implements 接口名表] {
```

```
    枚举常量定义
    [枚举体定义]
}
```

1. 枚举声明

被声明为 public 的枚举类型，可被其他包中的类访问，否则只能在定义它的包中使用。关键字 enum 指明当前定义的枚举类型，而不是类或接口。枚举类型与类一样，也可以实现接口。所有的枚举类型都隐含地继承了 java.lang.Enum 类，由于 Java 不支持多继承，所以枚举类型的声明中不能再继承任何类。

2. 枚举常量定义

枚举类型实际上是具有固定实例的特殊类。这些固定的实例就是通过枚举常量定义的。枚举常量是枚举类型的 static，final 的实例，是枚举类型的“值”，因此枚举常量可以在其他程序中通过枚举类型名进行引用，如对例 5-17 中枚举 Week 的常量引用 Week.SUNDAY。

最简单的枚举类型只包含一组枚举常量，如例 5-17 所示。枚举常量定义格式为：

```
常量1[,常量2[,…常量n]][;]
```

枚举常量之间用“,”分隔，最后用“;”表示结束。如果没有枚举体定义部分，则“;”可省略。每个枚举常量都将与一个整数值相对应，第一个枚举常量值为 0，第二个为 1，以此类推。

枚举类型是特殊的类。枚举类型的一个常量，实际上就是枚举类型的一个实例。在虚拟机加载枚举类型时，会调用枚举类型的构造方法，创建各个枚举实例。与类的构造方法一样，如果在枚举类型中没有显式地定义构造方法，编译器将自动为其提供一个默认构造方法。如果枚举类型在枚举体内定义了自己的构造方法，则在定义枚举常量时，可采用：

```
常量(参数1,参数2,…)
```

的形式，则在创建该枚举实例时将按照参数列表调用相应的构造方法。如果枚举常量使用默认的或枚举体内定义的不带参数的构造方法，则“()”可以省略，如例 5-17 所示。

3. 枚举体的定义

枚举体的定义与类的定义一样，可以包含变量、构造方法与成员方法，且定义形式也和类一样。但枚举类型的构造方法只能定义为 private，默认也为 private，这保证除了系统创建的枚举常量外，不会有任何其他程序调用枚举类型构造方法创建新的实例。

5.7.3 枚举类型的方法

Java 中，所有的枚举类型都默认继承于 java.lang.Enum 类。由于 java.lang.Enum 直接继承 java.lang.Object 且实现了 java.lang.Comparable 接口，所以每个枚举类型都具有 Object 类和 Comparable 接口中可被继承的方法，常用的方法包括如下几种。

- final Boolean equals(Object other)

如果 other 指向的对象等于此枚举常量时，返回 true。

- final String name()

返回此枚举常量的名称。

- final int ordinal()

返回此枚举常量在枚举类型定义中的位置序数,第一个常量的序数值为 0。

- String name()

返回枚举常量的名称,该名称与枚举常量在枚举类型声明中的名称完全一样。

- String toString()

返回枚举常量包含在枚举类型声明中的名称。该方法可以被重写,以返回枚举常量的其他名称。

- Static<T extends Enum<T>>T valueOf(class<T>enumType,String name)

返回指定枚举类型中指定名称的枚举常量。

另外,编译器在创建一个枚举时也将自动加入一些特殊的方法。例如,编译器将加入一个静态方法 values(),该方法返回一个包含该枚举类型所有常量的数组,并且数组中常量的顺序与枚举类型中声明的顺序相同。values()方法经常和 for 循环一起使用,实现对一个枚举类型所有值(常量)的遍历。例 5-18 中,利用 values()方法打印输出枚举 Week 中的所有常量。

例 5-18 枚举类型的 values()方法使用示例。

```
enum Week {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
public class EnumValuesTest{
    public static void main(String args[]){
        for (Week w:Week.values()){
            System.out.print(w.name() + ", ");
        }
        System.out.println();
    }
}
```

例 5-18 的运行结果为:

```
SUNDAY. MONDAY. TUESDAY. WEDNESDAY. THURSDAY. FRIDAY. SATURDAY.
```

5.7.4 枚举的使用

枚举类型可以像其他类型一样使用,可以定义数组,可以作为参数类型。枚举类型的变量也属于一种引用型变量,还可以通过枚举常量引用枚举类型中定义的成员。

枚举类型还可以和 switch 语句结合使用,如例 5-19 所示。例 5-19 中,定义了一个枚举类型 Coin 和一个最简单的枚举类型 CoinColor。Coin 中定义了 PENNY, NICKEL 等 5 个表示 5 种硬币枚举常量;定义了一个变量 value 表示硬币的面值,构造方法 Coin(int value),以及一个普通方法 value()。这 5 个枚举常量在创建时,将调用所定义的构造方法,对私有变量 value 赋值。在 CoinTest 类的 main()方法中,调用 Coin. values()方法获得枚举 Coin 中的所有常量,并通过 for 循环和 switch 语句遍历这些常量,输出每个常量的名称、面值和颜色。

例 5-19 枚举类型使用示例。

```
enum Coin {
```

```
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);
    private final int value;
    Coin(int value) {
        this.value = value;
    }
    public int value() {
        return value;
    }
}
enum CoinColor { COPPER, NICKEL, SILVER }
public class CoinTest {
    public static void main(String[] args) {
        for (Coin c : Coin.values()){
            System.out.print(c + ": " + c.value() + ", ");
            switch(c) {
                case PENNY:
                    System.out.println(CoinColor.COPPER);
                    break;
                case NICKEL:
                    System.out.println(CoinColor.NICKEL);
                    break;
                case DIME:
                case QUARTER:
                    System.out.println(CoinColor.SILVER);
                    break;
            }
        }
    }
}
```

例 5-19 的运行结果如下：

```
PENNY: 1, COPPER
NICKEL: 5, NICKEL
DIME: 10, SILVER
QUARTER: 25, SILVER
```

5.8　包装类与自动装箱和拆箱

5.8.1　基本数据类型的包装类

Java 中，从执行效率的角度考虑，基本数据类型如 int，double 等不作为类来对待。同时 Java 提供了 Wrapper 类即包装类用来把基本数据类型表示成类。每个 Java 基本数据类型在 java.lang 包中都有一个对应的 Wrapper 类，如表 5-2 所示。每个 Wrapper 类对象都封装了基本类型的一个值。数值型类型的包装类包括 Byte，Short，Integer，Long，Float 和 Double 都是抽象类 java.lang.Number 的子类。

表 5-2 Java 基本数据类的包装类

基本数据类型	Wrapper 类	基本数据类型	Wrapper 类
boolean	Boolean	int	Integer
byte	Byte	long	Long
char	Character	float	Float
short	Short	double	Double

把基本数据类型的一个值传递给包装类的相应构造函数,可以构造 Wrapper 类的对象。例如:

```
int pint = 500;
Integer wInt = new Integer(pInt);
int p2 = wInt.intValue();
```

Wrapper 类中包含了很多有用的方法和常量。如数值型 Wrapper 类中的 MIN_VALUE 和 MAX_VALUE 常量,定义了该类型的最小值与最大值。ByteValue(),shortValue()方法进行数值转换,valueOf()和 toString()实现字符串与数值之间的转换。例如:

```
int x = Integer.valueOf(str).intValue();
int y = Integer.parseInt(str);
```

5.8.2 自动装箱和拆箱

从 JDK 1.5 开始,Java 对基本类型的数据提供了自动装箱(autoboxing)和自动拆箱(autounboxing)功能。当编译器发现程序在应该使用对象的地方使用了基本数据类型的数据,编译器将把该数据包装为该基本类型对应的包装类的对象,这称为自动装箱。类似地,当编译器发现在应该使用基本类型数据的地方了包装类的对象,则会把该对象拆箱,从中取出所包含的基本类型数据,这称为自动拆箱。例 5-20 给出了一个整型数值自动装箱与拆箱的示例。

例 5-20 自动装箱与拆箱示例。

```
public class AutoBoxingTest{
    public static void main(String args[]){
        Integer x, y;
        int c;
        //自动装箱,将 x,y 构造为两个 Integer 对象
        x = 22;
        y = 15;

        if ( (c = x.compareTo(y)) = = 0)
            System.out.println("x is equal to y");
        else
            if ( c < 0 )
                System.out.println("x is less than y");
            else
```

```
            System.out.println("x is greater than y");

        System.out.println("The sum of x and y is " + (x + y) );
    }
}
```

例 5-20 的运行结果如下：

```
x is greater than y
The sum of x and y is 37
```

在例 5-20 中，变量 x 和 y 赋予了整型值。但由于它们是 Integer 类型的，编译器将自动把它们封装为两个 Integer 对象，这使得下面的 x.compareTo(y)方法能正常运行。在程序最后的 println()方法中，要进行 x 和 y 所包含整型值的加法运算，所以 x 和 y 将被自动拆箱，这样它们才能进行整数加法。

5.9　小　　结

本章介绍了 Java 的高级特征，包括 static 变量、static 方法与 static 语句块、抽象类与接口、包、泛型与集合类、枚举类型、包装类与自动装箱和拆箱。其中抽象类、接口与 package 是 Java 面向对象的重要高级特征，也是本章的重点。抽象类的主要作用是建立一类对象的抽象模型，定义通用的接口，支持多态。接口可以认为是一种极度抽象的抽象类，利用接口实现多重继承，避免了程序的复杂性与不安全性，使代码简单、可靠。package 机制体现了 Java 的封装特性，为 Java 类的管理、访问控制、命名管理等提供了有效的方法。泛型提供了编译时的类型安全保证，增加了程序的可读性和强壮性。集合类是 Java 中一组很实用的类，应该掌握 Java 集合类的框架与各种集合类的特点与用法，在使用集合类时要尽量使用泛型。枚举是 Java 新增加的表示一组常量的一种类型，一个枚举类型相当于一个类，具有很强大的功能。

习　题　5

1. 举例说明类方法与实例方法以及类变量与实例变量之间的区别。
2. 什么是接口？接口的意义是什么？
3. 什么是包？如何定义包？
4. 什么是抽象类？抽象类与接口有何区别？
5. 下列接口的定义中，哪个是正确的？

(1)
```
interface Printable{
    void print(){ };
    }
```

(2)
```
abstract interface Printable{
    void print();
    }
```

(3)
```
abstract interface Printable extends Interface1,Interface2{
    void print(){ };
    }
```

(4)
```
interface Printable{
    void print();
    }
```

6. 在一个图书管理程序中,类 Book,Newspaper 和 Video 都是类 Media 的子类。编写一个类,该类能够实现对一组书、报纸等的存储,并提供一定的检索功能。

7. 利用枚举类型重新编写例 5-13。