第14章

集合类

（视频讲解：13分钟）

集合可以看作是一个容器，如红色的衣服可以看作是一个集合，所有 Java 类的书也可以看作是一个集合。对于集合中的各个对象很容易将其存放到集合中，也很容易将其从集合中取出来，还可以将其按照一定的顺序进行摆放。Java 中提供了不同的集合类，这些类具有不同的存储对象的方式，并提供了相应的方法以方便用户对集合进行遍历、添加、删除以及查找指定的对象。学习 Java 语言一定要学会使用集合。本章将介绍 Java 中的各种集合类。

通过阅读本章，您可以：

- 了解集合类的概念
- 掌握 Collection 接口
- 掌握 List 集合
- 掌握 Set 集合
- 掌握 Map 集合

14.1 集合类概述

视频讲解：光盘\TM\lx\14\集合类概述.exe

java.util 包中提供了一些集合类，这些集合类又被称为容器。提到容器不难想到数组，集合类与数组的不同之处是，数组的长度是固定的，集合的长度是可变的；数组用来存放基本类型的数据，集合用来存放对象的引用。常用的集合有 List 集合、Set 集合和 Map 集合，其中 List 与 Set 继承了 Collection 接口，各接口还提供了不同的实现类。上述集合类的继承关系如图 14.1 所示。

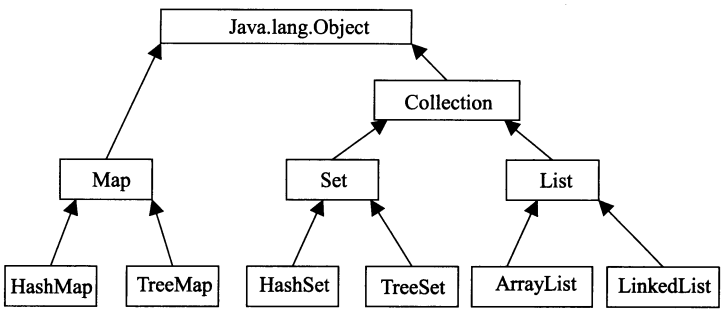

图 14.1 常用集合类的继承关系

14.2 Collection 接口

视频讲解：光盘\TM\lx\14\Collection 接口.exe

Collection 接口是层次结构中的根接口。构成 Collection 的单位称为元素。Collection 接口通常不能直接使用，但该接口提供了添加元素、删除元素、管理数据的方法。由于 List 接口与 Set 接口都继承了 Collection 接口，因此这些方法对 List 集合与 Set 集合是通用的。常用方法如表 14.1 所示。

表 14.1 Collection 接口的常用方法

方 法	功 能 描 述
add(E e)	将指定的对象添加到该集合中
remove(Object o)	将指定的对象从该集合中移除
isEmpty()	返回 boolean 值，用于判断当前集合是否为空
iterator()	返回在此 Collection 的元素上进行迭代的迭代器。用于遍历集合中的对象
size()	返回 int 型值，获取该集合中元素的个数

如何遍历集合中的每个元素呢？通常遍历集合，都是通过迭代器（Iterator）来实现。Collection 接口中的 iterator()方法可返回在此 Collection 进行迭代的迭代器。下面的实例就是典型的遍历集合的方法。

【例 14.1】 在项目中创建类 Muster，在主方法中实例化集合对象，并向集合中添加元素，最后将集合中的对象以 String 形式输出。（**实例位置：光盘\TM\sl\14.01**）

```
import java.util.*;                                    //导入 java.util 包，其他实例都要添加该语句
public class Muster {                                  //创建类 Muster
    public static void main(String args[]) {
        Collection<String> list = new ArrayList<>();   //实例化集合类对象
        list.add("a");                                 //向集合添加数据
        list.add("b");
        list.add("c");
        Iterator<String> it = list.iterator();         //创建迭代器
        while (it.hasNext()) {                         //判断是否有下一个元素
            String str = (String) it.next();           //获取集合中元素
            System.out.println(str);
        }
    }
}
```

运行结果如图 14.2 所示。

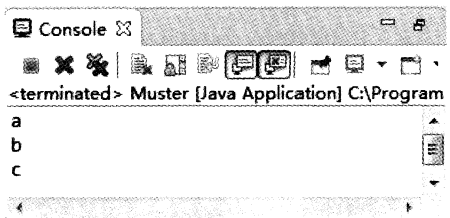

图 14.2　例 14.1 的运行结果

注意

Iterator 的 next()方法返回的是 Object。

14.3　List 集合

List 集合包括 List 接口以及 List 接口的所有实现类。List 集合中的元素允许重复，各元素的顺序就是对象插入的顺序。类似 Java 数组，用户可通过使用索引（元素在集合中的位置）来访问集合中的元素。

14.3.1　List 接口

视频讲解：光盘\TM\lx\14\List 接口.exe

List 接口继承了 Collection 接口，因此包含 Collection 中的所有方法。此外，List 接口还定义了以下两个非常重要的方法。

☑ get(int index)：获得指定索引位置的元素。

☑ set(int index,Object obj)：将集合中指定索引位置的对象修改为指定的对象。

14.3.2 List 接口的实现类

视频讲解：光盘\TM\lx\14\List 接口的实现类.exe

List 接口的常用实现类有 ArrayList 与 LinkedList。

☑ ArrayList 类实现了可变的数组，允许保存所有元素，包括 null，并可以根据索引位置对集合进行快速的随机访问；缺点是向指定的索引位置插入对象或删除对象的速度较慢。

☑ LinkedList 类采用链表结构保存对象。这种结构的优点是便于向集合中插入和删除对象，需要向集合中插入、删除对象时，使用 LinkedList 类实现的 List 集合的效率较高；但对于随机访问集合中的对象，使用 LinkedList 类实现 List 集合的效率较低。

使用 List 集合时通常声明为 List 类型，可通过不同的实现类来实例化集合。

【例 14.2】 分别通过 ArrayList、LinkedList 类实例化 List 集合，代码如下：

```
List<E> list = new ArrayList<>();
List<E> list2 = new LinkedList<>();
```

在上面的代码中，E 可以是合法的 Java 数据类型。例如，如果集合中的元素为字符串类型，那么 E 可以修改为 String。

【例 14.3】 在项目中创建类 Gather，在主方法中创建集合对象，通过 Math 类的 random()方法随机获取集合中的某个元素，然后移除数组中索引位置是“2”的元素，最后遍历数组。（**实例位置：光盘\TM\sl\14.02**）

```
public class Gather {                                    //创建类 Gather
    public static void main(String[] args) {             //主方法
        List<String> list = new ArrayList<>();           //创建集合对象
        list.add("a");                                   //向集合添加元素
        list.add("b");
        list.add("c");
        int i = (int) (Math.random()*list.size());       //获得 0~2 之间的随机数
        System.out.println("随机获取数组中的元素：" + list.get(i));
        list.remove(2);                                  //将指定索引位置的元素从集合中移除
        System.out.println("将索引是'2'的元素从数组移除后，数组中的元素是：");
        for (int j = 0; j < list.size(); j++) {          //循环遍历集合
            System.out.println(list.get(j));
        }
    }
}
```

运行结果如图 14.3 所示。Math 类的 random()方法可获得一个 0.0~1.0 之间的随机数。

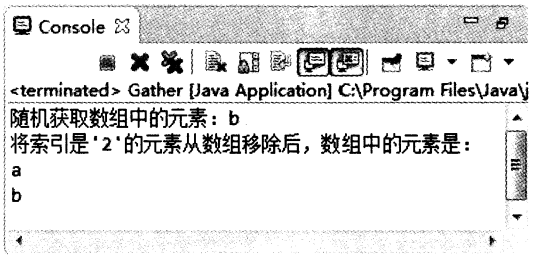

图 14.3　例 14.3 的运行结果

说明

与数组相同，集合的索引也是从 0 开始。

14.4　Set 集合

视频讲解：光盘\TM\lx\14\Set 集合.exe

Set 集合中的对象不按特定的方式排序，只是简单地把对象加入集合中，但 Set 集合中不能包含重复对象。Set 集合由 Set 接口和 Set 接口的实现类组成。Set 接口继承了 Collection 接口，因此包含 Collection 接口的所有方法。

注意

Set 的构造有一个约束条件，传入的 Collection 对象不能有重复值，必须小心操作可变对象（Mutable Object）。如果一个 Set 中的可变元素改变了自身状态导致 Object.equals(Object)=true，则会出现一些问题。

Set 接口常用的实现类有 HashSet 类与 TreeSet 类。

- ☑ HashSet 类实现 Set 接口，由哈希表（实际上是一个 HashMap 实例）支持。它不保证 Set 的迭代顺序，特别是它不保证该顺序恒久不变。此类允许使用 null 元素。
- ☑ TreeSet 类不仅实现了 Set 接口，还实现了 java.util.SortedSet 接口，因此，TreeSet 类实现的 Set 集合在遍历集合时按照自然顺序递增排序，也可以按照指定比较器递增排序，即可以通过比较器对用 TreeSet 类实现的 Set 集合中的对象进行排序。TreeSet 类新增的方法如表 14.2 所示。

表 14.2　TreeSet 类增加的方法

方　法	功 能 描 述
first()	返回此 Set 中当前第一个（最低）元素
last()	返回此 Set 中当前最后一个（最高）元素
comparator()	返回对此 Set 中的元素进行排序的比较器。如果此 Set 使用自然顺序，则返回 null

续表

方　法	功 能 描 述
headSet(E toElement)	返回一个新的 Set 集合，新集合是 toElement（不包含）之前的所有对象
subSet(E fromElement, E fromElement)	返回一个新的 Set 集合，是 fromElement（包含）对象与 fromElement（不包含）对象之间的所有对象
tailSet(E fromElement)	返回一个新的 Set 集合，新集合包含对象 fromElement（包含）之后的所有对象

【例 14.4】 在项目中创建类 UpdateStu，实现 Comparable 接口，重写该接口中的 compareTo()方法。在主方法中创建 UpdateStu 对象，创建集合，并将 UpdateStu 对象添加到集合中。遍历该集合中的全部元素，以及通过 headSet()、subSet()方法获得的部分集合。（**实例位置：光盘\TM\sl\14.03**）

```
import java.util.Iterator;
import java.util.TreeSet;

public class UpdateStu implements Comparable<Object> {          //创建类实现 Comparable 接口
    String name;
    long id;

    public UpdateStu(String name, long id) {                    //构造方法
        this.id = id;
        this.name = name;
    }

    public int compareTo(Object o) {
        UpdateStu upstu = (UpdateStu) o;
        int result = id > upstu.id ? 1 : (id == upstu.id ? 0 : -1);     //参照代码说明
        return result;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public static void main(String[] args) {
        UpdateStu stu1 = new UpdateStu("李同学", 01011);
        UpdateStu stu2 = new UpdateStu("陈同学", 01021);        //创建 UpdateStu 对象
```

```
        UpdateStu stu3 = new UpdateStu("王同学", 01051);
        UpdateStu stu4 = new UpdateStu("马同学", 01012);
        TreeSet<UpdateStu> tree = new TreeSet<>();
        tree.add(stu1);                                    //向集合添加对象
        tree.add(stu2);
        tree.add(stu3);
        tree.add(stu4);
        Iterator<UpdateStu> it = tree.iterator();     //Set 集合中的所有对象的迭代器
        System.out.println("Set 集合中的所有元素：");
        while (it.hasNext()) {                             //遍历集合
            UpdateStu stu = (UpdateStu) it.next();
            System.out.println(stu.getId() + " " + stu.getName());
        }
        it = tree.headSet(stu2).iterator();                //截取排在 stu2 对象之前的对象
        System.out.println("截取前面部分的集合：");
        while (it.hasNext()) {                             //遍历集合
            UpdateStu stu = (UpdateStu) it.next();
            System.out.println(stu.getId() + " " + stu.getName());
        }
        it = tree.subSet(stu2, stu3).iterator();           //截取排在 stu2 与 stu3 之间的对象
        System.out.println("截取中间部分的集合");
        while (it.hasNext()) {                             //遍历集合
            UpdateStu stu = (UpdateStu) it.next();
            System.out.println(stu.getId() + " " + stu.getName());
        }
    }
}
```

运行结果如图 14.4 所示。

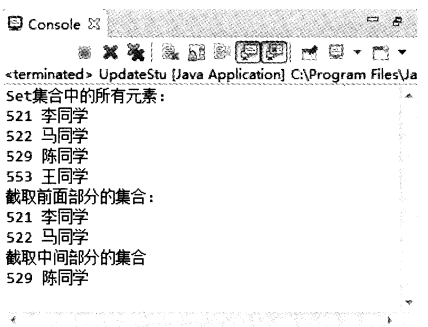

图 14.4　例 14.4 的运行结果

代码说明：存入 TreeSet 类实现的 Set 集合必须实现 Comparable 接口，该接口中的 compareTo(Object o) 方法比较此对象与指定对象的顺序。如果该对象小于、等于或大于指定对象，则分别返回负整数、0 或正整数。

技巧

headSet()、subSet()、tailSet()方法截取对象生成新集合时是否包含指定的参数，可通过如下方法来判别：如果指定参数位于新集合的起始位置，则包含该对象，如 subSet()方法的第一个参数和 tailSet()方法的参数；如果指定参数是新集合的终止位置，则不包含该参数，如 headSet()方法的入口参数和 subSet()方法的第二个入口参数。

14.5 Map 集合

Map 集合没有继承 Collection 接口，其提供的是 key 到 value 的映射。Map 中不能包含相同的 key，每个 key 只能映射一个 value。key 还决定了存储对象在映射中的存储位置，但不是由 key 对象本身决定的，而是通过一种“散列技术”进行处理，产生一个散列码的整数值。散列码通常用作一个偏移量，该偏移量对应分配给映射的内存区域的起始位置，从而确定存储对象在映射中的存储位置。Map 集合包括 Map 接口以及 Map 接口的所有实现类。

14.5.1 Map 接口

视频讲解：光盘\TM\lx\14\Map 接口.exe

Map 接口提供了将 key 映射到值的对象。一个映射不能包含重复的 key，每个 key 最多只能映射到一个值。Map 接口中同样提供了集合的常用方法，除此之外还包括如表 14.3 所示的常用方法。

表 14.3 Map 接口中的常用方法

方　法	功 能 描 述
put(K key, V value)	向集合中添加指定的 key 与 value 的映射关系
containsKey(Object key)	如果此映射包含指定 key 的映射关系，则返回 true
containsValue(Object value)	如果此映射将一个或多个 key 映射到指定值，则返回 true
get(Object key)	如果存在指定的 key 对象，则返回该对象对应的值，否则返回 null
keySet()	返回该集合中的所有 key 对象形成的 Set 集合
values()	返回该集合中所有值对象形成的 Collection 集合

下面通过实例介绍 Map 接口中某些方法的使用。

【例 14.5】 在项目中创建类 UpdateStu，在主方法中创建 Map 集合，并获取 Map 集合中所有 key 对象的集合和所有 values 值的集合，最后遍历集合。（**实例位置：光盘\TM\sl\14.04**）

```
public class UpdateStu {
    public static void main(String[] args) {
        Map<String,String> map = new HashMap<>();          //创建 Map 实例
        map.put("01", "李同学");                            //向集合中添加对象
        map.put("02", "魏同学");
```

```
        Set <String> set = map.keySet();                    //构建 Map 集合中所有 key 对象的集合
        Iterator <String> it = set.iterator();              //创建集合迭代器
        System.out.println("key 集合中的元素：");
        while (it.hasNext()) {                              //遍历集合
            System.out.println(it.next());
        }
        Collection <String> coll = map.values();            //构建 Map 集合中所有 values 值的集合
        it = coll.iterator();
        System.out.println("values 集合中的元素：");
        while (it.hasNext()) {                              //遍历集合
            System.out.println(it.next());
        }
    }
}
```

运行结果如图 14.5 所示。

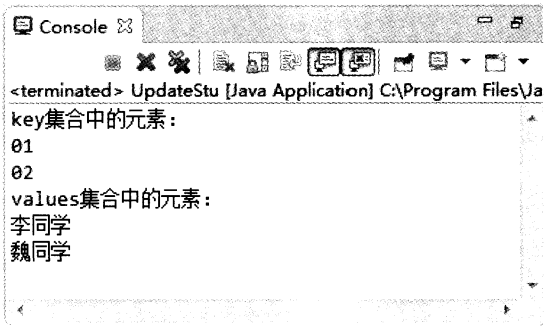

图 14.5　例 14.5 的运行结果

说明

Map 集合中允许值对象是 null，而且没有个数限制。例如，可通过“map.put("05",null);”语句向集合中添加对象。

14.5.2　Map 接口的实现类

视频讲解：光盘\TM\lx\14\Map 接口的实现类.exe

Map 接口常用的实现类有 HashMap 和 TreeMap。建议使用 HashMap 类实现 Map 集合，因为由 HashMap 类实现的 Map 集合添加和删除映射关系效率更高。HashMap 是基于哈希表的 Map 接口的实现，HashMap 通过哈希码对其内部的映射关系进行快速查找；而 TreeMap 中的映射关系存在一定的顺序，如果希望 Map 集合中的对象也存在一定的顺序，应该使用 TreeMap 类实现 Map 集合。

- ☑ HashMap 类是基于哈希表的 Map 接口的实现，此实现提供所有可选的映射操作，并允许使用 null 值和 null 键，但必须保证键的唯一性。HashMap 通过哈希表对其内部的映射关系进行快速查找。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。
- ☑ TreeMap 类不仅实现了 Map 接口，还实现了 java.util.SortedMap 接口，因此，集合中的映射关系具有一定的顺序。但在添加、删除和定位映射关系时，TreeMap 类比 HashMap 类性能稍差。由于 TreeMap 类实现的 Map 集合中的映射关系是根据键对象按照一定的顺序排列的，因此不允许键对象是 null。

可以通过 HashMap 类创建 Map 集合，当需要顺序输出时，再创建一个完成相同映射关系的 TreeMap 类实例。

【例 14.6】 通过 HashMap 类实例化 Map 集合，并遍历该 Map 集合，然后创建 TreeMap 实例实现将集合中的元素顺序输出。（**实例位置：光盘\TM\sl\14.05**）

（1）首先创建 Emp 类，代码如下：

```
public class Emp {
    private String e_id;
    private String e_name;
    public Emp( String e_id,String e_name) {
        this.e_id = e_id;
        this.e_name = e_name;
    }
    /*************省略了属性的 setXXX()以及 getXXX()方法******************/
}
```

（2）创建一个用于测试的主类。首先新建一个 Map 集合，并添加集合对象。分别遍历由 HashMap 类与 TreeMap 类实现的 Map 集合，观察两者的不同点。关键代码如下：

```
public class MapText {                                          //创建类 MapText
    public static void main(String[] args) {                   //主方法
        Map<String,String> map = new HashMap<>();              //由 HashMap 实现的 Map 对象

        Emp emp = new Emp("351", "张三");                       //创建 Emp 对象
        Emp emp2 = new Emp("512", "李四");
        Emp emp3 = new Emp("853", "王一");
        Emp emp4 = new Emp("125", "赵六");
        Emp emp5 = new Emp("341", "黄七");

        map.put(emp4.getE_id(), emp4.getE_name());              //将对象添加到集合中
        map.put(emp5.getE_id(), emp5.getE_name());
        map.put(emp.getE_id(), emp.getE_name());
        map.put(emp2.getE_id(), emp2.getE_name());
        map.put(emp3.getE_id(), emp3.getE_name());

                Set <String> set = map.keySet();                //获取 Map 集合中的 key 对象集合
        Iterator <String> it = set.iterator();
        System.out.println("HashMap 类实现的 Map 集合，无序：");
        while (it.hasNext()) {
```

```
            String str = (String) it.next();
            String name = (String) map.get(str);                    //遍历 Map 集合
            System.out.println(str + " " + name);
        }
        TreeMap<String,String> treemap = new TreeMap<>();           //创建 TreeMap 集合对象
        treemap.putAll(map);                                        //向集合添加对象
        Iterator <String> iter = treemap.keySet().iterator();
        System.out.println("TreeMap 类实现的 Map 集合，键对象升序：");
        while (iter.hasNext()) {                                    //遍历 TreeMap 集合对象
            String str = (String) iter.next();                      //获取集合中的所有 key 对象
            String name = (String) treemap.get(str);                //获取集合中的所有 values 值
            System.out.println(str + " " + name);
        }
    }
}
```

运行结果如图 14.6 所示。

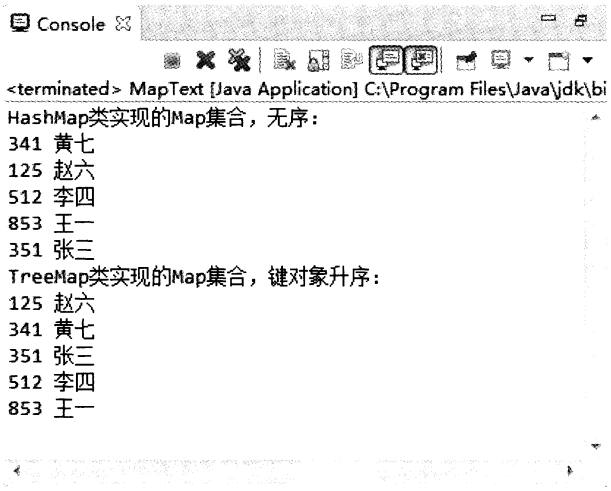

图 14.6　例 14.6 的运行结果

14.6　小　　结

本章介绍了 Java 中常见的集合，包括 List 集合、Set 集合和 Map 集合。对于每种集合的特点应该有所了解，重点掌握集合的遍历、添加对象、删除对象的方法。本章在介绍每种集合时都给出了典型、实用的小例子，以帮助读者掌握集合类的常用方法。集合是 Java 语言中很重要的部分，通过本章的学习，读者应该学会使用集合类。

14.7 实践与练习

1．将 1~100 之间的所有正整数存放在一个 List 集合中，并将集合中索引位置是 10 的对象从集合中移除。（**答案位置：光盘\TM\sl\14.06**）

2．分别向 Set 集合以及 List 集合中添加“A”“a”“c”“C”“a”5 个元素，观察重复值“a”能否重复地在 List 集合以及 Set 集合中添加。（**答案位置：光盘\TM\sl\14.07**）

3．创建 Map 集合，创建 Emp 对象，并将 Emp 对象添加到集合中（Emp 对象的 id 作为 Map 集合的键），并将 id 为“015”的对象从集合中移除。（**答案位置：光盘\TM\sl\14.08**）

第17章

枚举类型与泛型

（视频讲解：20分钟）

枚举类型可以取代以往常量的定义方式，即将常量封装在类或接口中，此外，它还提供了安全检查功能。枚举类型本质上还是以类的形式存在。泛型的出现不仅可以让程序员少写某些代码，主要的作用是解决类型安全问题，它提供编译时的安全检查，不会因为将对象置于某个容器中而失去其类型。本章将着重讲解枚举类型与泛型。

通过阅读本章，您可以：

- 掌握枚举类型
- 掌握泛型

17.1　枚 举 类 型

使用枚举类型可以取代以往定义常量的方式，同时枚举类型还赋予程序在编译时进行检查的功能。本节就来详细介绍枚举类型。

17.1.1　使用枚举类型设置常量

视频讲解：光盘\TM\lx\17\使用枚举类型设置常量.exe

以往设置常量，通常将常量放置在接口中，这样在程序中就可以直接使用，并且该常量不能被修改，因为在接口中定义常量时，该常量的修饰符为 final 与 static。常规定义常量的代码如例 17.1 所示。

【例 17.1】 在项目中创建 Constants 接口，在接口中定义常量的常规方式。

```
public interface Constants {
    public static final int Constants_A=1;
    public static final int Constants_B=12;
}
```

枚举类型出现后，逐渐取代了这种常量定义方式。使用枚举类型定义常量的语法如下：

```
public enum Constants{
    Constants_A,
    Constants_B,
    Constants_C
}
```

其中，enum 是定义枚举类型关键字。当需要在程序中使用该常量时，可以使用 Constants.Constants_A 来表示。

下面举例介绍枚举类型定义常量的方式。

【例 17.2】 在项目中创建 Constants 接口，在该接口中定义两个整型变量，其修饰符都是 static 和 final；之后定义名称为 Constants2 的枚举类，将 Constants 接口的常量放置在该枚举类中；最后，创建名称为 Constants 的类文件，在该类中通过 doit()和 doit2()进行不同方式的调用，然后再通过主方法进行调用，体现枚举类型定义常量的方式。（**实例位置：光盘\TM\sl\17.01**）

```
interface Constants {                                    //将常量放置在接口中
   public static final int Constants_A = 1;
   public static final int Constants_B = 12;
}
public class ConstantsTest {
   enum Constants2 {                                     //将常量放置在枚举类型中
        Constants_A, Constants_B
   }
```

```
    //使用接口定义常量
    public static void doit(int c) {                    //定义一个方法，这里的参数为 int 型
        switch (c) {                                    //根据常量的值做不同操作
            case Constants.Constants_A:
                System.out.println("doit() Constants_A");
                break;
            case Constants.Constants_B:
                System.out.println("doit() Constants_B");
                break;
        }
    }
    public static void doit2(Constants2 c) {            //定义一个参数对象是枚举类型的方法
        switch (c) {                                    //根据枚举类型对象做不同操作
            case Constants_A:
                System.out.println("doit2() Constants_A");
                break;
            case Constants_B:
                System.out.println("doit2() Constants_B");
                break;
        }
    }
    public static void main(String[] args) {
        ConstantsTest.doit(Constants.Constants_A);      //使用接口中定义的常量
        ConstantsTest.doit2(Constants2.Constants_A);    //使用枚举类型中的常量
        ConstantsTest.doit2(Constants2.Constants_B);    //使用枚举类型中的常量
        ConstantsTest.doit(3);
        //ConstantsTest.doit2(3);
    }
}
```

在 Eclipse 中运行本实例，运行结果如图 17.1 所示。

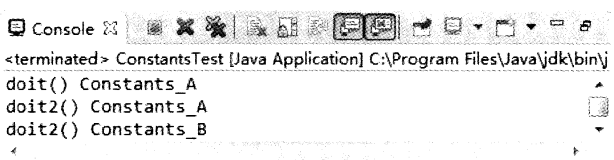

图 17.1　使用枚举类型定义常量

在上述代码中，当用户调用 doit()方法时，即使编译器不接受在接口中定义的常量参数，也不会报错；但调用 doit2()方法，任意传递参数，编译器就会报错，因为这个方法只接受枚举类型的常量作为其参数。

枚举类型也可以在类的内部进行定义，下面将介绍如何在类的内部进行枚举类型的定义。

【例 17.3】 在项目中创建 ConstantsTest 类，该类中以内部类的形式定义枚举类型。

```
public class ConstantsTest {
    enum Constants2{ //将常量放置在枚举类型中
        Constants_A,
        Constants_B
```

```
    }
...
}
```

这种形式类似于内部类形式，当编译该类时，除了 ConstantsTest.class 外，还存在 ConstantsTest$1.class 与 ConstantsTest$Constants2.class 文件。

17.1.2　深入了解枚举类型

视频讲解：光盘\TM\lx\17\深入了解枚举类型.exe

1．操作枚举类型成员的方法

枚举类型较传统定义常量的方式，除了具有参数类型检测的优势之外，还具有其他方面的优势。

用户可以将一个枚举类型看作是一个类，它继承于 java.lang.Enum 类，当定义一个枚举类型时，每一个枚举类型成员都可以看作是枚举类型的一个实例，这些枚举类型成员都默认被 final、public、static 修饰，所以当使用枚举类型成员时直接使用枚举类型名称调用枚举类型成员即可。

由于枚举类型对象继承于 java.lang.Enum 类，所以该类中一些操作枚举类型的方法都可以应用到枚举类型中。表 17.1 中列举了枚举类型中的常用方法。

表 17.1　枚举类型的常用方法

方 法 名 称	具 体 含 义	使 用 方 法	举　例
values()	该方法可以将枚举类型成员以数组的形式返回	枚举类型名称.values()	Constants2.values()
valueOf()	该方法可以实现将普通字符串转换为枚举实例	枚举类型名称.valueOf("abc")	Constants2.valueOf("abc")
compareTo()	该方法用于比较两个枚举对象在定义时的顺序	枚举对象.compareTo()	Constants_A.compareTo(Constants_B)
ordinal()	该方法用于得到枚举成员的位置索引	枚举对象.ordinal()	Constants_A.ordinal()

（1）values()

枚举类型实例包含一个 values()方法，该方法将枚举类型的成员变量实例以数组的形式返回，也可以通过该方法获取枚举类型的成员。

【例 17.4】 在项目中创建 ShowEnum 类，在该类中使用枚举类型中的 values()方法获取枚举类型中的成员变量。（**实例位置：光盘\TM\sl\17.02**）

```
import static java.lang.System.out;
public class ShowEnum {
    enum Constants2 { //将常量放置在枚举类型中
        Constants_A, Constants_B
    }
    //循环由 values()方法返回的数组
    public static void main(String[] args) {
        for (int i = 0; i < Constants2.values().length; i++) {
```

```
            //将枚举成员变量打印
            out.println("枚举类型成员变量：" + Constants2.values()[i]);
        }
    }
}
```

在 Eclipse 中运行本实例，结果如图 17.2 所示。

在例 17.4 中，由于 values()方法将枚举类型的成员以数组的形式返回，所以根据该数组的长度进行循环操作，然后将该数组中的值返回。

（2）valueOf()与 compareTo()

枚举类型中静态方法 valueOf()可以将普通字符串转换为枚举类型，而 compareTo()方法用于比较两个枚举类型对象定义时的顺序。

【例 17.5】 在项目中创建 EnumMethodTest 类，在该类中使用枚举类型中的 valueOf()与 compareTo()方法。（**实例位置：光盘\TM\sl\17.03**）

```
import static java.lang.System.out;
public class EnumMethodTest {
    enum Constants2 { //将常量放置在枚举类型中
        Constants_A, Constants_B
    }
    //定义比较枚举类型方法，参数类型为枚举类型
    public static void compare(Constants2 c) {
        //根据 values()方法返回的数组做循环操作
        for (int i = 0; i < Constants2.values().length; i++) {
            //将比较结果返回
            out.println(c + "与" + Constants2.values()[i] + "的比较结果为："
                    + c.compareTo(Constants2.values()[i]));
        }
    }
    //在主方法中调用 compare()方法
    public static void main(String[] args) {
        compare(Constants2.valueOf("Constants_B"));
    }
}
```

在 Eclipse 中运行本实例，结果如图 17.3 所示。

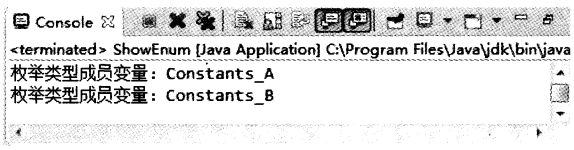

图 17.2　使用枚举类型中的 values()方法获取枚举类型中的成员变量

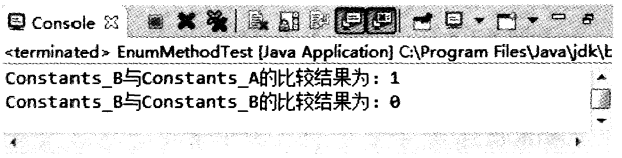

图 17.3　使用 compareTo()方法比较两个枚举类型成员定义的顺序

调用 compareTo()方法返回的结果，正值代表方法中参数在调用该方法的枚举对象位置之前；0 代表两个互相比较的枚举成员的位置相同；负值代表方法中参数在调用该方法的枚举对象位置之后。

（3）ordinal()

枚举类型中的 ordinal()方法用于获取某个枚举对象的位置索引值。

【例 17.6】 在项目中创建 EnumIndexTest 类，在该类中使用枚举类型中的 ordinal()方法获取枚举类型成员的位置索引。（**实例位置：光盘\TM\sl\17.04**）

```
import static java.lang.System.out;
public class EnumIndexTest {
    enum Constants2 { //将常量放置在枚举类型中
        Constants_A, Constants_B, Constants_C
    }
    public static void main(String[] args) {
        for (int i = 0; i < Constants2.values().length; i++) {
            //在循环中获取枚举类型成员的索引位置
            out.println(Constants2.values()[i] + "在枚举类型中位置索引值"
                    + Constants2.values()[i].ordinal());
        }
    }
}
```

在 Eclipse 中运行本实例，结果如图 17.4 所示。

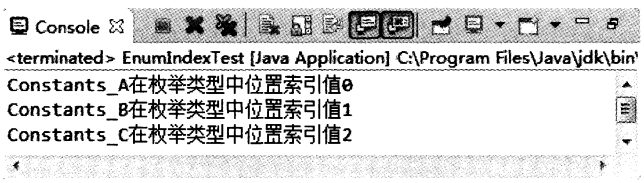

图 17.4　获取枚举类型成员的位置索引

在例 17.6 中，在循环中获取每个枚举对象时，调用 ordinal()方法即可相应获取该枚举类型成员的索引位置。

2．枚举类型中的构造方法

在枚举类型中，可以添加构造方法，但是规定这个构造方法必须为 private 修饰符所修饰。枚举类型定义的构造方法语法如下：

```
enum 枚举类型名称{
    Constants_A("我是枚举成员 A"),
    Constants_B("我是枚举成员 B"),
    Constants_C("我是枚举成员 C"),
    Constants_D(3);
    private String description;
    private Constants2(){            //定义默认构造方法
    }
    //定义带参数的构造方法，参数类型为字符串型
    private Constants2(String description) {
        this.description=description;
    }
    private Constants2(int i){       //定义带参数的构造方法，参数类型为整型
```

```
        this.i=this.i+i;
    }
}
```

从枚举类型构造方法的语法中可以看出，无论是无参构造方法还是有参构造方法，修饰权限都为 private。定义一个有参构造方法后，需要对枚举类型成员相应地使用该构造方法，如 Constants_A("我是枚举成员 A")和 Constants_D(3)语句，相应地使用了参数为 String 型和参数为 int 型的构造方法。然后可以在枚举类型中定义两个成员变量，在构造方法中为这两个成员变量赋值，这样就可以在枚举类型中定义该成员变量的 getXXX()方法了。

下面是在枚举类型中定义构造方法的实例。

【例 17.7】 在项目中创建 EnumIndexTest 类，在该类中定义枚举类型的构造方法。（**实例位置：光盘\TM\sl\17.05**）

```
import static java.lang.System.out;
public class EnumIndexTest {
    enum Constants2 {                                    //将常量放置在枚举类型中
        Constants_A("我是枚举成员 A"),                    //定义带参数的枚举类型成员
        Constants_B("我是枚举成员 B"),
        Constants_C("我是枚举成员 C"),
        Constants_D(3);
        private String description;
        private int i = 4;
        private Constants2() {
        }
         //定义参数为 String 型的构造方法
        private Constants2(String description) {
            this.description = description;
        }
        private Constants2(int i) {                      //定义参数为 int 型的构造方法
            this.i = this.i + i;
        }
        public String getDescription() {                 //获取 description 的值
            return description;
        }
        public int getI() {                              //获取 i 的值
            return i;
        }
    }
    public static void main(String[] args) {
        for (int i = 0; i < Constants2.values().length; i++) {
            out.println(Constants2.values()[i]+"调用 getDescription()方法为："
                    + Constants2.values()[i].getDescription());
        }
        out.println(Constants2.valueOf("Constants_D") + "调用 getI()方法为："
                + Constants2.valueOf("Constants_D").getI());
    }
}
```

在 Eclipse 中运行本实例，结果如图 17.5 所示。

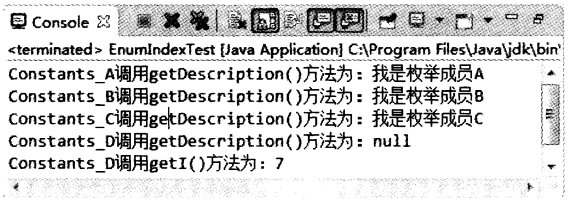

图 17.5　在枚举类型中定义构造方法

在本实例中，调用 getDescription()和 getI()方法，返回在枚举类型定义的构造方法中设置的操作。这里将枚举类型中的构造方法设置为 private 修饰，以防止客户代码实例化一个枚举对象。

除了可以使用例 17.7 中所示的方式定义 getDescription()方法获取枚举类型成员定义时的描述之外，还可以将这个 getDescription()方法放置在接口中，使枚举类型实现该接口，然后使每个枚举类型实现接口中的方法。

【例 17.8】 在项目中创建 d 接口和枚举类型的 AnyEnum 类，在枚举类型 AnyEnum 类中实现带方法的接口，使每个枚举类型成员实现该接口中的方法。（**实例位置：光盘\TM\sl\17.06**）

```
import static java.lang.System.out;
interface d {
    public String getDescription();
    public int getI();
}
public enum AnyEnum implements d {
    Constants_A { //可以在枚举类型成员内部设置方法
        public String getDescription() {
            return ("我是枚举成员 A");
        }
        public int getI() {
            return i;
        }
    },
    Constants_B {
        public String getDescription() {
            return ("我是枚举成员 B");
        }
        public int getI() {
            return i;
        }
    },
    Constants_C {
        public String getDescription() {
            return ("我是枚举成员 C");
        }
        public int getI() {
            return i;
        }
    },
    Constants_D {
        public String getDescription() {
            return ("我是枚举成员 D");
```

```
        }
        public int getI() {
            return i;
        }
    };
    private static int i = 5;
    public static void main(String[] args) {
        for (int i = 0; i < AnyEnum.values().length; i++) {
            out.println(AnyEnum.values()[i] + "调用 getDescription()方法为："
                    + AnyEnum.values()[i].getDescription());
            out.println(AnyEnum.values()[i] + "调用 getI()方法为："
                    + AnyEnum.values()[i].getI());
        }
    }
}
```

在 Eclipse 中运行本实例，结果如图 17.6 所示。

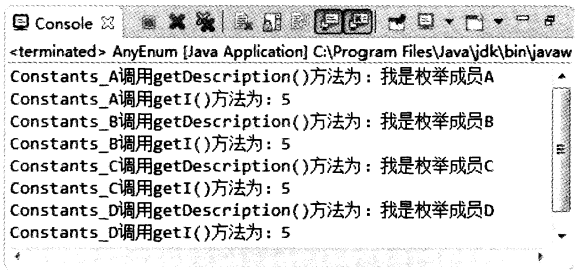

图 17.6　在每个枚举类型成员中实现接口中的方法

17.1.3　使用枚举类型的优势

视频讲解：光盘\TM\lx\17\使用枚举类型的优势.exe

枚举类型声明提供了一种用户友好的变量定义方法，枚举了某种数据类型所有可能出现的值。总结枚举类型，它具有以下特点：

- ☑ 类型安全。
- ☑ 紧凑有效的数据定义。
- ☑ 可以和程序其他部分完美交互。
- ☑ 运行效率高。

17.2　泛　　型

泛型实质上就是使程序员定义安全的类型。在没有出现泛型之前，Java 也提供了对 Object 的引用“任意化”操作，这种“任意化”操作就是对 Object 引用进行向下转型及向上转型操作，但某些强制

类型转换的错误也许不会被编译器捕捉，而在运行后出现异常，可见强制类型转换存在安全隐患，所以在此提供了泛型机制。本节就来探讨泛型机制。

17.2.1　回顾向上转型与向下转型

视频讲解：光盘\TM\lx\17\回顾向上转型与向下转型.exe

在介绍泛型之前，先来看一个例子。

【例 17.9】 在项目中创建 Test 类，在该类中使基本类型向上转型为 Object 类型。（**实例位置：光盘\TM\sl\17.07**）

```
public class Test {
    private Object b;                          //定义 Object 类型成员变量
    public Object getB() {                     //设置相应的 getXXX()方法
        return b;
    }
    public void setB(Object b) {               //设置相应的 setXXX()方法
        this.b = b;
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.setB(new Boolean(true));             //向上转型操作
        System.out.println(t.getB());
        t.setB(new Float(12.3));
        Float f = (Float) (t.getB());          //向下转型操作
        System.out.println(f);
    }
}
```

运行本实例，结果如图 17.7 所示。

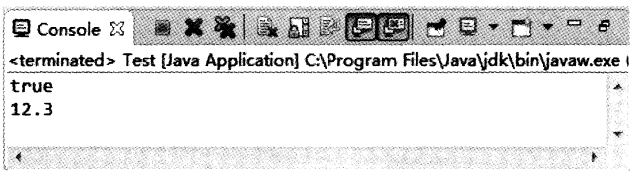

图 17.7　使基本类型向上转型为 Object 类型

在本实例中，Test 类中定义了私有的成员变量 b，它的类型为 Object 类型，同时为其定义了相应的 setXXX()与 getXXX()方法。在类主方法中，将 new Boolean(true)对象作为 setB()方法的参数，由于 setB()方法的参数类型为 Object，这样就实现了向上转型操作。同时在调用 getB()方法时，将 getB()方法返回的 Object 对象以相应的类型返回，这个就是向下转型操作，问题通常就会出现在这里。因为向上转型是安全的，而如果进行向下转型操作时用错了类型，或者并没有执行该操作，就会出现异常,例如以下代码：

```
t.setB(new Float(12.3));
Integer f=(Integer)(t.getB());
System.out.println(f);
```

并不存在语法错误，所以可以被编译器接受，但在执行时会出现 ClassCastException 异常。这样看来，向下转型操作通常会出现问题，而泛型机制有效地解决了这一问题。

17.2.2 定义泛型类

视频讲解：光盘\TM\lx\17\定义泛型类.exe

Object 类为最上层的父类，很多程序员为了使程序更为通用，设计程序时通常使传入的值与返回的值都以 Object 类型为主。当需要使用这些实例时，必须正确地将该实例转换为原来的类型，否则在运行时将会发生 ClassCastException 异常。

在 JDK 1.5 版本以后，提出了泛型机制。其语法如下：

```
类名<T>
```

其中，T 代表一个类型的名称。

如果将例 17.9 改写为定义类时使用泛型的形式，关键代码如例 17.10 所示。

【例 17.10】 在项目中创建 OverClass 类，该类定义了泛型类。

```
public class OverClass<T> {                    //定义泛型类
    private T over;                            //定义泛型成员变量
    public T getOver() {                       //设置 getXXX()方法
        return over;
    }
    public void setOver(T over) {              //设置 setXXX()方法
        this.over = over;
    }
    public static void main(String[] args) {
        //实例化一个 Boolean 型的对象
        OverClass<Boolean> over1 = new OverClass<Boolean>();
        //实例化一个 Float 型的对象
        OverClass<Float> over2 = new OverClass<Float>();
        over1.setOver(true);                   //不需要进行类型转换
        over2.setOver(12.3f);
        Boolean b = over1.getOver();           //不需要进行类型转换
        Float f = over2.getOver();
        System.out.println(b);
        System.out.println(f);
    }
}
```

运行上述代码，结果与图 17.7 所示的结果一致。在例 17.10 中定义类时，在类名后添加了一个<T>语句，这里便使用了泛型机制。可以将 OverClass 类称为泛型类，同时返回和接受的参数使用 T 这个类型。最后在主方法中可以使用 Over<Boolean>形式返回一个 Boolean 型的对象，使用 OverClass<Float>形式返回一个 Float 型的对象，使这两个对象分别调用 setOver()方法不需要进行显式向上转型操作，setOver()方法直接接受相应类型的参数，而调用 getOver()方法时，不需要进行向下转型操作，直接将 getOver()方法返回的值赋予相应的类型变量即可。

从例 17.10 中可以看出，使用泛型定义的类在声明该类对象时可以根据不同的需求指定<T>真正的类型，而在使用类中的方法传递或返回数据类型时将不再需要进行类型转换操作，而是使用在声明泛型类对象时“<>”符号中设置的数据类型。

使用泛型这种形式将不会发生 ClassCastException 异常，因为在编译器中就可以检查类型匹配是否正确。

【例 17.11】 在项目中定义泛型类。

```
OverClass<Float> over2=new OverClass<Float>();
over2.setOver(12.3f);
//Integer i=over2.getOver(); //不能将 boolean 型的值赋予 Integer 变量
```

在例 17.11 中，由于 over2 对象在实例化时已经指定类型为 Float，而最后一条语句却将该对象获取出的 Float 类型值赋予 Integer 类型，所以编译器会报错。而如果使用向下转型操作，就会在运行上述代码时发生异常。

说明

在定义泛型类时，一般类型名称使用 T 来表达，而容器的元素使用 E 来表达，具体的设置读者可以参看 JDK 5.0 以上版本的 API。

17.2.3　泛型的常规用法

视频讲解：光盘\TM\lx\17\泛型的常规用法.exe

1．定义泛型类时声明多个类型

在定义泛型类时，可以声明多个类型。语法如下：

```
MutiOverClass<T1,T2>
MutiOverClass:泛型类名称
```

其中，T1 和 T2 为可能被定义的类型。

这样在实例化指定类型的对象时就可以指定多个类型。例如：

```
MutiOverClass<Boolean,Float>=new MutiOverClass<Boolean,Float>();
```

2．定义泛型类时声明数组类型

定义泛型类时也可以声明数组类型，下面的实例中定义泛型时便声明了数组类型。

【例 17.12】 在项目中创建 ArrayClass 类，在该类中定义泛型类声明数组类型。（**实例位置：光盘\TM\sl\17.08**）

```
public class ArrayClass<T> {
    private T[] array;                          //定义泛型数组
    public void SetT(T[] array) {               //设置 SetXXX()方法为成员数组赋值
        this.array = array;
    }
```

```
    public T[] getT() {                          //获取成员数组
        return array;
    }
    public static void main(String[] args) {
        ArrayClass<String> a = new ArrayClass<String>();
        String[] array = { "成员 1", "成员 2", "成员 3", "成员 4", "成员 5" };
        a.SetT(array);                           //调用 SetT()方法
        for (int i = 0; i < a.getT().length; i++) {
            System.out.println(a.getT()[i]);  //调用 getT()方法返回数组中的值
        }
    }
}
```

在 Eclipse 中运行本实例，结果如图 17.8 所示。

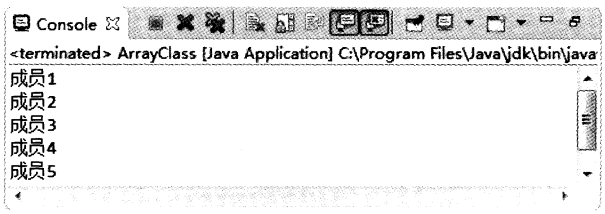

图 17.8　定义泛型类时声明数组类型

本实例在定义泛型类时声明一个成员数组，数组的类型为泛型，然后在泛型类中相应设置 setXXX()与 getXXX()方法。

可见，可以在使用泛型机制时声明一个数组，但是不可以使用泛型来建立数组的实例。例如，下面的代码就是错误的：

```
public class ArrayClass <T>{
    //private T[] array=new T[10]; //不能使用泛型来建立数组的实例
    ...
}
```

3．集合类声明容器的元素

可以使用 K 和 V 两个字符代表容器中的键值和与键值相对应的具体值。

【例 17.13】 在项目中创建 MutiOverClass 类，在该类中使用集合类声明容器的元素。（**实例位置：光盘\TM\sl\17.09**）

```
import java.util.HashMap;
import java.util.Map;
public class MutiOverClass<K, V> {
    public Map<K, V> m = new HashMap<K, V>();        //定义一个集合 HashMap 实例
     //设置 put()方法，将对应的键值与键名存入集合对象中
    public void put(K k, V v) {
        m.put(k, v);
    }
    public V get(K k) {                              //根据键名获取键值
```

```
        return m.get(k);
    }
    public static void main(String[] args) {
        //实例化泛型类对象
        MutiOverClass<Integer, String> mu
        = new MutiOverClass<Integer, String>();
        for (int i = 0; i < 5; i++) {
            //根据集合的长度循环将键名与具体值放入集合中
            mu.put(i, "我是集合成员" + i);
        }
        for (int i = 0; i < mu.m.size(); i++) {
            //调用 get()方法获取集合中的值
            System.out.println(mu.get(i));
        }
    }
}
```

在 Eclipse 中运行本实例，结果如图 17.9 所示。

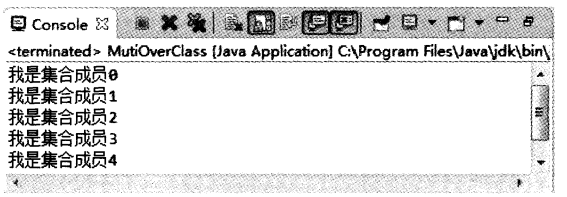

图 17.9　集合类声明容器的元素

其实在例 17.13 中定义的泛型类 MutiOverClass 纯属多余，因为在 Java 中这些集合框架已经都被泛型化了，可以在主方法中直接使用“public Map<K,V> m=new HashMap<K,V>();”语句创建实例，然后相应调用 Map 接口中的 put()与 get()方法完成填充容器或根据键名获取集合中具体值的功能。集合中除了 HashMap 这种集合类型之外，还包括 ArrayList、Vector 等。表 17.2 列举了几个常用的被泛型化的集合类。

表 17.2　常用的被泛型化的集合类

集　合　类	泛 型 定 义
ArrayList	ArrayList<E>
HashMap	HashMap<K,V>
HashSet	HashSet<E>
Vector	Vector<E>

下面的实例演示了这些集合的使用方式。

【例 17.14】 在项目中创建 AnyClass 类，在该类中使用泛型实例化常用集合类。（**实例位置：光盘\TM\sl\17.10**）

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import java.util.Vector;
public class AnyClass {
```

```
    public static void main(String[] args) {
        //定义 ArrayList 容器，设置容器内的值类型为 Integer
        ArrayList<Integer> a = new ArrayList<Integer>();
        a.add(1);                                    //为容器添加新值
        for (int i = 0; i < a.size(); i++) {
            //根据容器的长度循环显示容器内的值
            System.out.println("获取 ArrayList 容器的值：" + a.get(i));
        }
        //定义 HashMap 容器，设置容器的键名与键值类型分别为 Integer 与 String 型
        Map<Integer, String> m = new HashMap<Integer, String>();
        for (int i = 0; i < 5; i++) {
            m.put(i, "成员" + i);                    //为容器填充键名与键值
        }
        for (int i = 0; i < m.size(); i++) {
            //根据键名获取键值
            System.out.println("获取 Map 容器的值" + m.get(i));
        }
        //定义 Vector 容器，使容器中的内容为 String 型
        Vector<String> v = new Vector<String>();
        for (int i = 0; i < 5; i++) {
            v.addElement("成员" + i);            //为 Vector 容器添加内容
        }
        for (int i = 0; i < v.size(); i++) {
            //显示容器中的内容
            System.out.println("获取 Vector 容器的值" + v.get(i));
        }
    }
}
```

在 Eclipse 中运行本实例，结果如图 17.10 所示。

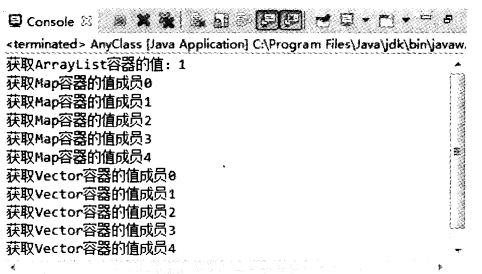

图 17.10　使用泛型实例化常用集合类

17.2.4　泛型的高级用法

视频讲解：光盘\TM\lx\17\泛型的高级用法.exe

泛型的高级用法包括限制泛型可用类型和使用类型通配符等。

1．限制泛型可用类型

默认可以使用任何类型来实例化一个泛型类对象，但 Java 中也对泛型类实例的类型作了限制。语法如下：

```
class 类名称<T extends anyClass>
```

其中，anyClass 指某个接口或类。

使用泛型限制后，泛型类的类型必须实现或继承了 anyClass 这个接口或类。无论 anyClass 是接口还是类，在进行泛型限制时都必须使用 extends 关键字。

【例 17.15】 在项目中创建 LimitClass 类，在该类中限制泛型类型。

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
public class LimitClass<T extends List> { //限制泛型的类型
    public static void main(String[] args) {
        //可以实例化已经实现 List 接口的类
        LimitClass<ArrayList> l1 = new LimitClass<ArrayList>();
        LimitClass<LinkedList> l2 = new LimitClass<LinkedList>();
        //这句是错误的，因为 HashMap 没有实现 List()接口
        //LimitClass<HashMap> l3=new LimitClass<HashMap>();
    }
}
```

在例 17.15 中，将泛型作了限制，设置泛型类型必须实现 List 接口。例如，ArrayList 和 LinkedList 都实现了 List 接口，而 HashMap 没有实现 List 接口，所以在这里不能实例化 HashMap 类型的泛型对象。

当没有使用 extends 关键字限制泛型类型时，默认 Object 类下的所有子类都可以实例化泛型类对象。如图 17.11 所示的两个语句是等价的。

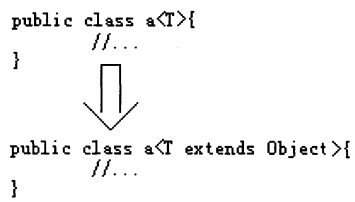

图 17.11　两个等价的泛型类

2．使用类型通配符

在泛型机制中，提供了类型通配符，其主要作用是在创建一个泛型类对象时限制这个泛型类的类型实现或继承某个接口或类的子类。要声明这样一个对象可以使用“?”通配符来表示，同时使用 extends 关键字来对泛型加以限制。

使用泛型类型通配符的语法如下：

```
泛型类名称<? extends List> a=null;
```

其中，<? extends List>表示类型未知，当需要使用该泛型对象时，可以单独实例化。

【例 17.16】 在项目中创建一个类文件，在该类中限制泛型类型。

```
A<? extends List> a=null;
a=new A<ArrayList>();
a=new A<LinkedList>();
```

如果实例化没有实现 List 接口的泛型对象，编译器将会报错。例如，实例化 HashMap 对象时，编

译器将会报错，因为 HashMap 类没有实现 List 接口。

除了可以实例化一个限制泛型类型的实例之外，还可以将该实例放置在方法的参数中。

【例 17.17】 在项目中创建一个类文件，在该类中的方法参数中使用匹配字符串。

```
public void doSomething(A<? extends List> a){
}
```

在上述代码中，定义方式有效地限制了传入 doSomething()方法的参数类型。

如果使用 A<?>这种形式实例化泛型类对象，则默认表示可以将 A 指定为实例化 Object 及以下的子类类型。读者可能对这种编码类型有些疑惑，例 17.18 将直观地介绍 A<?>泛型机制。

【例 17.18】 在泛型中使用通配符形式。

```
List<String> l1=new ArrayList<String>();        //实例化一个 ArrayList 对象
l1.add("成员");                                  //在集合中添加内容
List<?> l2=l1;                                   //使用通配符
List<?> l3=new LinkedList<Integer>();
System.out.println(l2.get(0));                   //获取集合中第一个值
```

在例 17.18 中，List<?>类型的对象可以接受 String 类型的 ArrayList 集合，也可以接受 Integer 类型的 LinkedList 集合。也许有的读者会有疑问，List<?> l2=l1 语句与 List l2=l1 存在何种本质区别？这里需要注意的是，使用通配符声明的名称实例化的对象不能对其加入新的信息，只能获取或删除。例如：

```
l1.set(0, "成员改变");                           //没有使用通配符的对象调用 set()方法
//l2.set(0, "成员改变");                         //使用通配符的对象调用 set()方法，不能被调用
//l3.set(0, 1);
l2.get(0);                                       //可以使用 l2 的实例获取集合中的值
l2.remove(0);                                    //根据键名删除集合中的值
```

从上述代码中可以看出，由于对象 l1 是没有使用 A<?>这种形式初始化出来的对象，所以它可以调用 set()方法改变集合中的值，但 l2 与 l3 则是通过使用通配符的方式创建出来的，所以不能改变集合中的值。

技巧

泛型类型限制除了可以向下限制之外，还可以进行向上限制，只要在定义时使用 super 关键字即可。例如，“A<? super List> a=null;”这样定义后，对象 a 只接受 List 接口或上层父类类型，如“a=new A<Object>();”。

3. 继承泛型类与实现泛型接口

定义为泛型的类和接口也可以被继承与实现。

【例 17.19】 在项目中创建一个类文件，在该类中继承泛型类。

```
public class ExtendClass<T1>{
}
class SubClass<T1,T2,T3> extends ExtendClass<T1>{
}
```

如果在 SubClass 类继承 ExtendClass 类时保留父类的泛型类型，需要在继承时指明，如果没有指明，直接使用 extends ExtendsClass 语句进行继承操作，则 SubClass 类中的 T1、T2 和 T3 都会自动变为 Object，所以在一般情况下都将父类的泛型类型保留。

定义的泛型接口也可以被实现。

【例 17.20】 在项目中创建一个类文件，在该类中实现泛型接口。

```
interface i<T1>{
}
class SubClass2<T1,T2,T3> implements i<T1>{
}
```

17.2.5 泛型总结

视频讲解：光盘\TM\lx\17\泛型总结.exe

下面总结一下泛型的使用方法。

☑ 泛型的类型参数只能是类类型，不可以是简单类型，如 A<int>这种泛型定义就是错误的。

☑ 泛型的类型个数可以是多个。

☑ 可以使用 extends 关键字限制泛型的类型。

☑ 可以使用通配符限制泛型的类型。

17.3 小　　结

本章主要讲述了枚举类型以及泛型的用法。虽然枚举类型与泛型的语法比较简单，但是展开后的写法比较复杂，所以初学者应该仔细揣摩，并且对这两种机制做到简单掌握。此外，读者应该积极了解每个 JDK 版本新增的内容，而查看相应版本的 API 便是一种极为有效的手段。

17.4 实践与练习

1．尝试定义一个枚举类型类，使用 switch 语句获取枚举类型的值。（**答案位置：光盘\TM\sl\17.11**）

2．尝试定义一个泛型类，使用 extends 关键字限制该泛型类的类型为 List 接口，并分别创建两个泛型对象。（**答案位置：光盘\TM\sl\17.12**）

3．尝试定义一个泛型类，并使用通配符。（**答案位置：光盘\TM\sl\17.13**）