

Chapter 17 Java Data Structures

Objectives

- Understand the limitations of arrays.
- Become familiar with the Java Collections Framework hierarchy.
- Use the Iterator interface to traverse a collection.
- Know the Set interface and know when to use HashSet or TreeSet to store elements.
- Comprehend the List interface and know whether to use ArrayList or LinkedList to store elements.
- Understand the differences between Vector and ArrayList, and know how to use Vector and Stack.
- Understand the differences between Collection and Map, and know how to use Map to store values associated with the keys.
- Use the static methods in the Collections and Arrays class.

Introduction

In Chapter 7, "Arrays," you learned to store and process elements in arrays. Arrays can be used to store a group of primitive type values or a group of objects. Once an array is created, its size cannot be altered. Array is a useful data structure to represent a collection of elements, but it provides inadequate support for inserting, deleting, sorting, and searching operations. The Java 2 platform introduced several new interfaces and classes that can be used to organize and manipulate data efficiently. These new interfaces and classes are known as *Java Collections Framework*.

A *collection* is an object that represents a group of objects, often referred to as *elements*. The Java Collections Framework supports two types of collections, named *collections* and *maps*. They are defined in the interfaces Collection and Map. An instance of Collection represents a group of objects. An instance of Map represents a group of objects, each of which is associated with a key. You can get the object from a map using a key. So you have to put the object into the map using the key. A collection can be a set

or a list, defined in the interfaces Set and List, which are subinterfaces of Collection. An instance of Set stores a group of non-duplicate elements. An instance of List is an ordered collection of elements. The relationships of the interfaces and classes in the Java Collections Framework are shown in Figure 17.1 and Figure 17.2. These interfaces and classes provide a unified API for storing and processing a collection of objects efficiently. You will learn to use these interfaces and classes in this chapter.

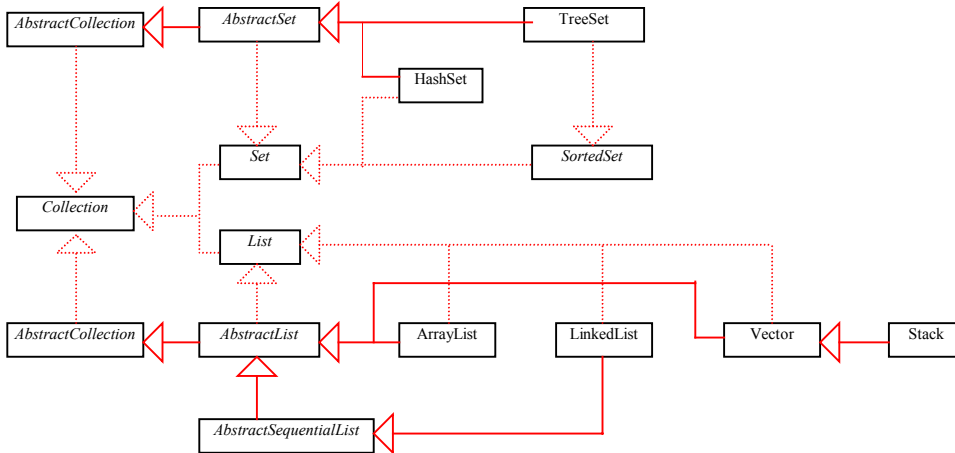


Figure 17.1

An instance of Collection stores a group of objects.

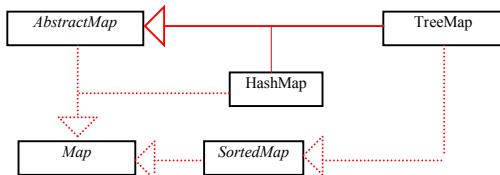


Figure 17.2

An instance of Map stores a group of objects with associated keys.

NOTE: All interfaces and classes defined in the Java Collections Framework are grouped in the java.util package.

The Collection Interface and the AbstractCollection Class

The Collection interface is the root interface for storing and processing a collection of objects. Its public methods are listed in Figure 17.3.

<i>Collection</i>
<code>+add(Object element): boolean</code>
<code>+addAll(Collection collection): boolean</code>
<code>+clear(): void</code>
<code>+contains(Object element): boolean</code>
<code>+containsAll(Collection collection): boolean</code>
<code>+equals(Object object): boolean</code>
<code>+hashCode(): int</code>
<code>+iterator(): Iterator</code>
<code>+remove(Object element): boolean</code>
<code>+removeAll(Collection collection): boolean</code>
<code>+retainAll(Collection collection): boolean</code>
<code>+size(): int</code>
<code>+toArray(): Object[]</code>
<code>+toArray(Object[] array): Object[]</code>

Figure 17.3

The Collection interface is the root for the Set and List interfaces.

The Collection interface provides the basic operations for adding and removing elements in a collection. The add method adds an element to the collection. The addAll method adds the specified collection to this collection. The remove method removes an element from the collection, if present. The removeAll method removes the elements from this collection that are present in the specified collection. The retainAll method retains the elements in this collection that are also present in the specified collection. All these methods return boolean. The return value is true if the collection is changed as a result of the method execution. The clear() method simply removes all the elements from the collection.

The Collection interface provides the query operations. The size method returns the number of elements in the collection. The contains method checks if the collection contains the specified element. The containsAll method checks if the collection contains all the elements in the specified collection. The isEmpty method returns true if the collection is empty.

The Collection interface provides two overloaded the methods to convert the collection into an array. The toArray() method returns an array representation for the collection.

The toArray(Object[] a) method returns an array with the specified runtime type.

The iterator method in the Collection interface returns an instance of Iterator, which can be used to traverse the collection using the next() method. You can also use the hasNext() method to check if there are more elements in the iterator, and use the remove() method to remove the last element returned by the iterator.

The AbstractCollection class is a convenience class that provides partial implementation for the Collection interface. It implements all the methods in Collection except the equals, hashCode, size and the iterator methods. These methods are implemented in appropriate subclasses.

NOTE: Some of the methods in the Collection interface are optional. If a concrete subclass does not support an optional method, the method should throw java.lang.UnsupportedOperationException. Since this exception is a subclass of RuntimeException, you don't have to place the method in a try-catch block.

The Set Interface, and the AbstractSet, and HashSet Classes

The Set interface extends the Collection interface. It does not introduce new methods or constants, but it stipulates that an instance of Set contains no duplicate elements. The concrete classes that implement Set must ensure that no duplicate elements can be added to the set. That is no two elements e1 and e2 can be in the set such that e1.equals(e2) is true.

The AbstractSet class is a convenience class that extends AbstractCollection and implements Set. The AbstractSet class provides concrete implementations for the equals method and the hashCode method. The hash code of a set is the sum of the hash code of all the elements in the set. Since the size method and iterator method are not implemented in the AbstractSet class, AbstractSet is an abstract class.

The HashSet class is a concrete class that implements Set. It can be used to store duplicate-free elements. For efficiency, objects added to a hash set need to implement the hashCode method in a manner that properly disperses the hash code. Most of the classes in the Java API implement the hashCode method. For example the hashCode in the String class is implemented as follows:

```
public int hashCode()  
_ {
```

```

    int h = 0;
    int off = offset;
    char val[] = value;
    int len = count;

    for (int i = 0; i < len; i++)
        h = 31*h + val[off++];

    return h;
}

```

The `hashCode` in the `Integer` class simply returns its `int` value.

Example 17.1 Using HashSet and Iterator

This example creates a hash set filled with strings, and uses an iterator to traverse the elements in the list. The output of the program is shown in Figure 17.4.

```

package chapter17;

import java.util.*;

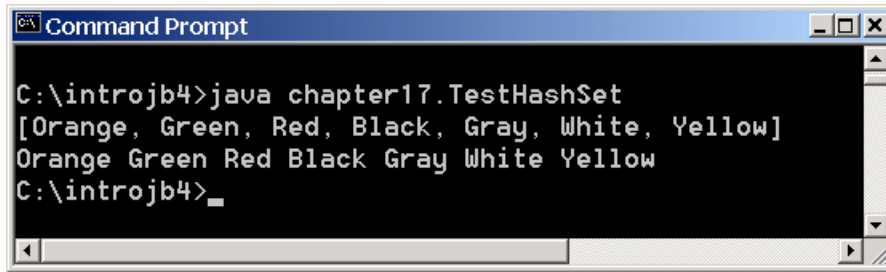
public class TestHashSet
{
    public static void main(String[] args)
    {
        // Create a hash set
        Set set = new HashSet();
        set.add("Red");
        set.add("Yellow");
        set.add("White");
        set.add("Green");
        set.add("Orange");
        set.add("Gray");
        set.add("Black");
        set.add("Red");

        System.out.println(set);

        // Obtain an iterator for the hash set
        Iterator iterator = set.iterator();

        // Display the elements in the hash set
        while (iterator.hasNext())
        {
            System.out.print(iterator.next() + " ");
        }
    }
}

```



```
Command Prompt
C:\introjb4>java chapter17.TestHashSet
[Orange, Green, Red, Black, Gray, White, Yellow]
Orange Green Red Black Gray White Yellow
C:\introjb4>
```

Figure 17.4

The program adds string elements to a hash set, displays the elements using the `toString` method, and traverses the elements using an iterator.

Example Review

The string "red" is added to the set twice, but only one is stored, because a set does not allow duplicates. Two objects `o1` and `o2` are duplicate, if `o1.equals(o2)` is true. So, it is important that the class of the objects overrides the `equals` method to compare the contents of the objects. The `equals` method is defined in the `Object` class. By default, it returns `o1 == o2`.

As shown in Figure 17.4, the strings are not stored in the order that they are added. There is no particular order for the elements in a hash set. To impose a specific order on the elements in a set, you need to use the `TreeSet` class, which is introduced in the next section.

NOTE: The iterator is fail-fast. That means if you are using an iterator to traverse a collection while the underlying collection is being modified by another thread, then the iterator fails immediately by throwing `java.util.ConcurrentModificationException`. Since this exception is a subclass of `RuntimeException`, you don't have to place the methods of `Iterator` in a try-catch block. In Exercise 17.1, you will write a program to create a situation that causes an iterator to throw `ConcurrentModificationException`.

The `SortedSet` Interface, the `TreeSet` Class, and the `Comparator` interface.

SortedSet is a subinterface of Set, which guarantees that the elements in the set are sorted. TreeSet is a concrete class that implements the SortedSet interface. You can use an iterator to traverse the elements in the sorted order. The elements can be sorted in two ways.

- One way is to use the Comparable interface. The objects added to the set are instances of Comparable, so they can be compared using the compareTo method. The Comparable interface was introduced in Chapter 6, "Class Inheritance." Several classes in the Java API, such as the String class, and all the wrapper classes for the primitive types, implement the Comparable interface. This approach is referred to as *natural order*,
- The other way is to specify a comparator for the elements in the set if the class for the elements does not implement the Comparable interface, or you don't want to use the compareTo method in the class that implements the Comparable interface. This approach is referred to as *order by comparator*.

If you choose to use a comparator, you have to create a class that implements the java.util.Comparator interface. The Comparator interface has two methods compare and equals.

```
[BL] public int compare(Object element1, Object element2)
```

This method returns a positive value, if element1 is less than element2; and returns a negative value, if element1 is greater than element2; and returns zero if they are equal.

```
[BL] public boolean equals(Object element)
```

This method returns true if the specified object is also a comparator and it imposes the same ordering as this comparator.

NOTE: The equals method is also defined in the Object class. Therefore, you will not get a compilation error, if you don't implement the equals method in your custom comparator class. However, implementing this method may, in some cases, improve performance by allowing programs to determine that two distinct comparators impose the same order.

For example, you can provide the following comparator to compare two elements of the GeometricObject class, defined in Section, "Abstract Classes," in Chapter 6, "Class Inheritance."

```

package chapter17;

import chapter6.GeometricObject;
import java.util.Comparator;

public class GeometricObjectComparator implements Comparator
{
    public int compare(Object o1, Object o2)
    {
        double area1 = ((GeometricObject)o1).findArea();
        double area2 = ((GeometricObject)o2).findArea();

        if (area1 < area2)
            return -1;
        else if (area1 == area2)
            return 0;
        else
            return 1;
    }
}

```

If you create a `TreeSet` using its default constructor, the `compareTo` method is used to compare the elements in the set, assuming that the class of the elements implements the `Comparable` interface. To use a comparator, you have to use the constructor `TreeSet(Comparator comparator)` to create a sorted set that uses the `compare` method in the comparator to order the elements in the set.

Example 17.2 Using `TreeSet` to Sort Elements in a Set

This example creates a hash set filled with strings, and then creates a tree set for the same strings. The strings are sorted in the tree set using the `compareTo` method in the `Comparable` interface. The example also creates a tree set of geometric objects. The geometric objects are sorted using the `compare` method in the `Comparator` interface. The output of the program is shown in Figure 17.5.

```

package chapter17;

import chapter6.*;
import java.util.*;

public class TestTreeSet
{
    public static void main(String[] args)
    {
        // Create a hash set of strings
        Set hashSet = new HashSet();
        hashSet.add("Red");
    }
}

```

```

    hashCode.add("Yellow");
    hashCode.add("White");
    hashCode.add("Green");
    hashCode.add("Orange");
    hashCode.add("Gray");
    hashCode.add("Black");

    System.out.println("An unsorted set of strings");
    System.out.println(hashCode + "\n");

    // Create a sorted tree set from the hash set for strings
    Set treeSet = new TreeSet(hashCode);
    System.out.println("A sorted set of strings");
    System.out.println(treeSet + "\n");

    // Create a tree set for geometric objects using a comparator
    Set geometricObjectSet =
        new TreeSet(new GeometricObjectComparator());
    geometricObjectSet.add(new Rectangle(4, 5));
    geometricObjectSet.add(new Circle(40));
    geometricObjectSet.add(new Circle(40));
    geometricObjectSet.add(new Cylinder(4, 1));

    // Obtain an iterator for the tree set of geometric objects
    Iterator iterator = geometricObjectSet.iterator();

    // Display geometric objects in the tree set
    System.out.println("A sorted set of geometric objects");
    while (iterator.hasNext())
    {
        GeometricObject object = (GeometricObject)iterator.next();
        System.out.println(object + ", area= " + object.findArea());
    }
}
}
}

```

```

C:\introjb4>java chapter17.TestTreeSet
An unsorted set of strings
[Orange, Green, Red, Black, Gray, White, Yellow]

A sorted set of strings
[Black, Gray, Green, Orange, Red, White, Yellow]

A sorted set of geometric objects
[Rectangle] width = 4.0 and height = 5.0, area= 20.0
[Cylinder] radius = 4.0 and length 1.0, area= 125.66370614359172
[Circle] radius = 40.0, area= 5026.548245743669

C:\introjb4>

```

Figure 17.5

The program demonstrates the differences between hash sets and tree sets.

Example Review

The circles are added to the set in the tree set, but only one is stored, because these two circles are equal and the set does not allow duplicates.

All the classes in Figure 17.1 have at least two constructors. One is the default constructor that constructs an empty collection. The other constructor constructs instances from a collection. So, the TreeSet class has a constructor TreeSet(Collection c) for constructing a TreeSet from a collection c. In this example, new TreeSet(hashSet) creates an instance of TreeSet from a collection hashSet.

TIP: If you don't need to maintain a sorted set when updating a set. You can use a hash set, because it is faster to insert and remove elements in a hash set. When you need a set to be sorted, you can convert the set into a tree set.

The List Interface, the AbstractList Class, and the AbstractSequentialList Class

A set stores non-duplicate elements. To allow duplicate elements to be stored in a collection, you need to use a list. A list can not only store duplicate elements, but also allows the user to specify where the element is stored. The user can access the element by index. The List interface extends Collection to define an ordered collection with duplicates allowed. The List interface adds position-oriented operations, as well as a new list iterator that enables the user to traverse the list bi-directionally. The new methods in the List interface are shown in Figure 17.6.

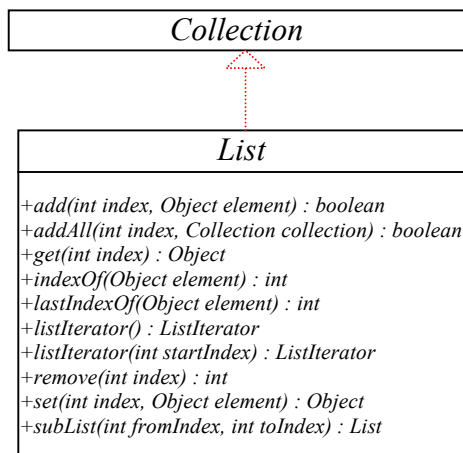


Figure 17.6

The *List* interface stores elements in sequence, permitting duplicates.

You can use the `add(index, element)` method to insert an element as a specified index, and use the `addAll(index, collection)` method to insert a collection at a specified index. You can use the `remove(index)` method to remove an element at the specified index from the list. You can set a new element at the specified index using the `set(index, element)` method.

You can obtain the index of the first occurrence of the specified element in the list using the `indexOf(element)` method, and obtain the index of the last occurrence of the specified element in the list using the `lastIndexOf(element)` method. You can obtain a sub-list using the method `subList(fromIndex, toIndex)`.

The method `listIterator()` or `listIterator(startIndex)` returns an instance of `ListIterator`. The `ListIterator` interface extends the `Iterator` interface to add bi-directional traversal of the list. The methods in `ListIterator` are listed in Figure 17.7.

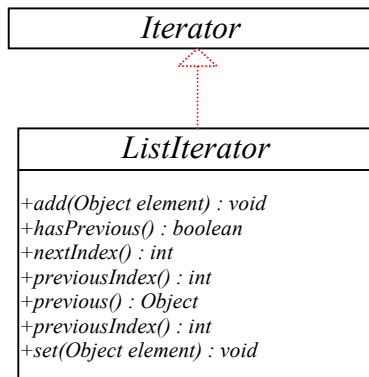


Figure 17.7

ListIterator enables traversal of a list bi-directionally.

The `add(element)` method inserts the specified element into the list. The element is inserted immediately before the next element that would be returned by the `next()` method defined in the `Iterator` interface, if any, and after the next element that would be returned by `previous`, if any. If the list contains no elements, the new element becomes the sole element on the list. You can use the `set(element)` method to replace the last element returned by the `next` method or the `previous` method with the specified element.

You can use the `hasNext()` method defined in the `List` interface to check if the iterator has more elements when traversed in the forward direction, and use the `hasPrevious()` method to check if the iterator has more elements when traversed in the backward direction.

The `next()` method defined in the `List` interface returns the next element in the iterator and the `previous()` method returns the previous element in the iterator. The `nextIndex()` method returns the index of the next element in the iterator, and the `previousIndex()` returns the index of the previous element in the iterator.

The `AbstractList` class provides a partial implementation for the `List` interface. The `AbstractSequentialList` class extends `AbstractList` to provide support for linked lists.

The `ArrayList` and `LinkedList` Classes

The `ArrayList` class and the `LinkedList` class are concrete implementations of the `List` interface. Which of the two classes you use depends on your specific needs. If you need to support random access through an index without inserting or removing elements from any place other than the end, `ArrayList` offers the most efficient collection. If, however,

your application requires the insertion or deletion of elements from any place in the list, you should choose LinkedList. A list can grow or shrink dynamically. An array is fixed once it is created. If your application does not require insertion or deletion of elements, the most efficient data structure is the array.

ArrayList is a resizable-array implementation of the List interface. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. Each ArrayList instance has a capacity. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added an ArrayList, its capacity grows automatically.

LinkedList is a linked list implementation of the List interface. In addition to implementing the List interface, this class provides the methods for retrieving, inserting, and removing elements from both ends of the list, as shown in Figure 17.8.

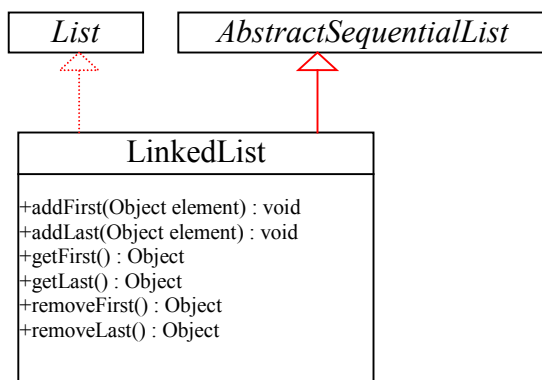


Figure 17.8

LinkedList provides methods for adding and inserting elements at both ends of the list.

Example 17.3 Using ArrayList and LinkedList

This example creates an array list filled with numbers, and inserts new elements into the specified location in the list. The example also creates a linked list from the array list, inserts and removes the elements from the list. Finally, the example traverses the list forward and backward. The output of the program is shown in Figure 17.9.

package chapter17;

```

import java.util.*;

public class TestList
{
    public static void main(String[] args)
    {
        ArrayList arrayList = new ArrayList();
        arrayList.add(new Integer(1));
        arrayList.add(new Integer(2));
        arrayList.add(new Integer(3));
        arrayList.add(new Integer(1));
        arrayList.add(new Integer(4));
        arrayList.add(0, new Integer(10));
        arrayList.add(3, new Integer(30));

        System.out.println("A list of integers in the array list:");
        System.out.println(arrayList);

        LinkedList linkedList = new LinkedList(arrayList);
        linkedList.add(1, "red");
        linkedList.removeLast();
        linkedList.addFirst("green");

        System.out.println("Display the linked list forawrd:");
        ListIterator listIterator = linkedList.listIterator();
        while (listIterator.hasNext())
        {
            System.out.print(listIterator.next() + " ");
        }
        System.out.println();

        System.out.println("Display the linked list backward:");
        listIterator = linkedList.listIterator(linkedList.size());
        while (listIterator.hasPrevious())
        {
            System.out.print(listIterator.previous() + " ");
        }
    }
}

```

```

C:\introjb4>java chapter17.TestList
A list of integers in the array list:
[10, 1, 2, 30, 3, 1, 4]
Display the linked list forawrd:
green 10 red 1 2 30 3 1
Display the linked list backwrdr:
1 3 30 2 1 red 10 green
C:\introjb4>

```

Figure 17.9

The program uses the array list, and linked lists.

Example Review

A list can hold identical elements. Integer 1 is stored twice in the list. ArrayList and LinkedList can be operated similarly. The critical differences between them are their internal implementation, which impacts the performance. ArrayList is efficient for retrieving elements, and for inserting and removing elements from the end of the list. LinkedList is efficient for inserting and removing elements anywhere in the list.

You can use TreeSet to store sorted elements. But there is no sorted list. However, the Java Collections Framework provides static methods in the Collections class that can be used to sort a list. The Collections class is introduced in the section, "The Collections Class," later in this chapter.

The Vector and the Stack Class

The Java Collections Framework was introduced with Java 2. Several data structures were supported prior to Java 2. Among them are the Vector class and the Stack class. These classes were redesigned to fit into the Java Collections Framework, but their old-style methods are retained for compatibility. This section introduces the Vector class and the Stack class.

In Java 2, Vector is the same as ArrayList, except that Vector contains the synchronized methods for accessing and modifying the vector. None of the new collection data structures introduced so far are synchronized. If synchronization is required, you can use the synchronized versions of the collection classes. These classes are introduced later in the section, "The Collections Class."

The Vector class implements the List interface. Additionally, it has the following methods contained in the original Vector class defined prior to Java 2, as shown in Figure 17.10.

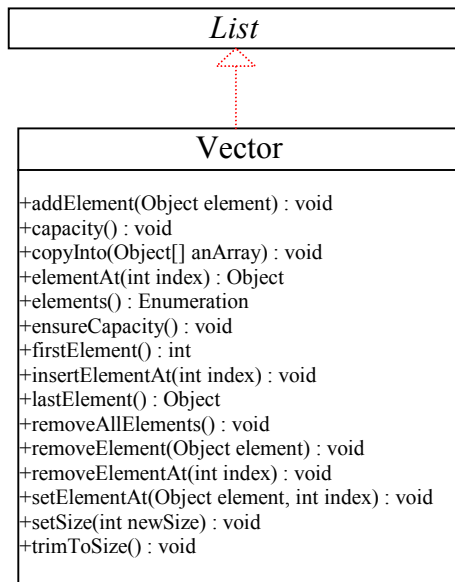


Figure 17.10

The Vector class in Java 2 implements List, but its old-style methods are retained.

Most of the additional methods in the Vector class listed in the previous UML diagram are similar to the methods in the List interface. These methods were introduced before the Java Collections Framework. For example, addElement(Object element) is the same as the add(Object element) method, except that addElement method is synchronized. You should use the ArrayList class if you don't need synchronization. ArrayList works much faster than Vector.

NOTE: The elements() method returns an Enumeration. The Enumeration interface was introduced prior to Java 2. This interface is superseded by the Iterator interface.

NOTE: Vector has been widely used in Java programming, because it was the Java resizable array implementation prior to Java 2. Many of the Swing data models use vectors. Swing data models are introduced in "Rapid Java Application Development Using JBuilder 4."

The Stack class represents a last-in-first-out stack of objects. The elements are accessed only from the top of the stack. You can retrieve, insert, or remove an element from the top of the stack. The Stack class extends the Vector class and provides the following additional methods, as shown in Figure 17.11.

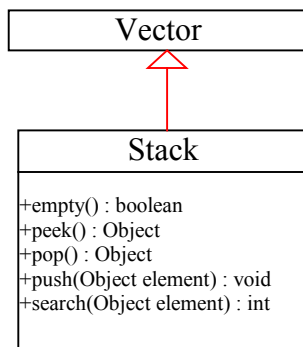


Figure 17.11

The Stack class extends Vector to provide a last-in-first-out data structure.

The Stack class was introduced prior to Java 2. The methods shown in the previous diagram were used before Java 2. The empty() method is the same as isEmpty(). The peek() method looks at the element at the top of the stack without removing it. The pop() method removes the top element from the stack, and returns the removed element. The push(Object element) method adds the specified element to the stack. The search(Object element) method checks if the specified element is in the stack.

Example 17.4 Using Vector and Stack

This example presents two programs to rewrite Example 7.1, using a vector and a stack instead of an array, respectively. The program reads student scores from the keyboard, stores the scores in the vector, finds the best scores, and then assigns grades for all the students. A negative score signals the end of the input. A sample run of the program is shown in Figure 7.12.

The first program that uses a vector is given as follows:

```

// AssignGradeUsingVector.java: Assign grade
package chapter17;

import java.util.*;
import chapter2.MyInput;

public class AssignGradeUsingVector
{
    // Main method
    public static void main(String[] args)
    {
        Vector scoreVector = new Vector(); // Vector to hold scores
  
```

```

    double best = 0; // The best score
    char grade; // The grade

    // Read scores and find the best score
    System.out.println("Please enter scores. " +
        "A negative score terminates input.");
    do
    {
        System.out.print("Please enter a new score: ");
        double score = MyInput.readDouble();

        if (score < 0) break;

        // Add the score into the vector
        scoreVector.addElement(new Double(score));

        // Find the best score
        if (score > best)
            best = score;
    } while (true);

    System.out.println("There are total " + scoreVector.size() +
        " students ");

    // Assign and display grades
    for (int i=0; i<scoreVector.size(); i++)
    {
        // Retrieve an element from the vector
        Double doubleObject = (Double)(scoreVector.elementAt(i));

        // Get the score
        double score = doubleObject.doubleValue();

        if (score >= best - 10)
            grade = 'A';
        else if (score >= best - 20)
            grade = 'B';
        else if (score >= best - 30)
            grade = 'C';
        else if (score >= best - 40)
            grade = 'D';
        else
            grade = 'F';

        System.out.println("Student " + i + " score is " + score +
            " and grade is " + grade);
    }
}

```

*****Insert Figure 17.12**

```
Command Prompt
C:\introjb4>java chapter17.AssignGradeUsingVector
Please enter scores. A negative score terminates input.
Please enter a new score: 70
Please enter a new score: 60
Please enter a new score: 50
Please enter a new score: 40
Please enter a new score: 0
Please enter a new score: -1
There are total 5 students
Student 0 score is 70.0 and grade is A
Student 1 score is 60.0 and grade is A
Student 2 score is 50.0 and grade is B
Student 3 score is 40.0 and grade is C
Student 4 score is 0.0 and grade is F
C:\introjb4>
```

Figure 17.12

The program receives scores, stores them in a vector, and assigns grades.

The previous program would work if you replace all the occurrences of Vector by Stack, because Stack is a subclass of Vector. However, you can rewrite the previous example exclusively using the stack operations that operate elements only from the top of the stack. The revision of the program using the Stack class is given as follows:

```
package chapter17;

import java.util.*;
import chapter2.MyInput;

public class AssignGradeUsingStack
{
    // Main method
    public static void main(String[] args)
    {
        Stack scoreStack = new Stack(); // Stack to hold scores
        double best = 0; // The best score
        char grade; // The grade

        // Read scores and find the best score
        System.out.println("Please enter scores. " +
            "A negative score terminates input.");
        do
        {
            System.out.print("Please enter a new score: ");
            double score = MyInput.readDouble();
```

```

_____ if (score < 0) break;

_____ // Add the score into the Stack
_____ scoreStack.push(new Double(score));

_____ // Find the best score
_____ if (score > best)
_____     best = score;
_____ } while (true);

_____ System.out.println("There are total " + scoreStack.size() +
_____     " students ");

_____ int i = scoreStack.size();

_____ // Assign and display grades
_____ while (!scoreStack.isEmpty())
_____ {
_____     // Retrieve an element from the Stack
_____     Double doubleObject = (Double)(scoreStack.pop());

_____     // Get the score
_____     double score = doubleObject.doubleValue();

_____     if (score >= best - 10)
_____         grade = 'A';
_____     else if (score >= best - 20)
_____         grade = 'B';
_____     else if (score >= best - 30)
_____         grade = 'C';
_____     else if (score >= best - 40)
_____         grade = 'D';
_____     else
_____         grade = 'F';

_____     System.out.println("Student " + i-- + " score is " + score +
_____         " and grade is " + grade);
_____ }
_____ }
_____ }

```

Example Review

Example 7.1 uses an array to store scores. The size of an array is fixed once the array is created. You should use array lists, linked lists, vectors, or stacks to store an unspecified number of elements.

The element type in an array can be primitive type values or objects, but the element type is the Java Collections Framework must be the Object type.

The Map Interface, the AbstractMap class, the SortedMap interface, the HashMap class, and the TreeMap class

The Collection interface represents a set of elements, stored in HashSet, TreeSet, ArrayList, LinkedList, Vector, or Stack. The Map interface maps keys to the elements. The keys are like indexes. In List, the indexes are integer. In Map, the keys can be any objects. A map cannot contain duplicate keys. Each key can map to at most one value. The Map interface provides the methods for querying, updating, and obtaining a collection of values and a set of keys, as shown in Figure 17.13.

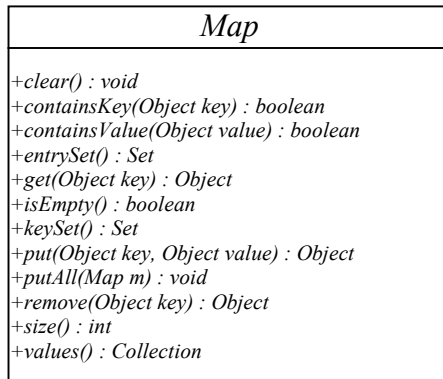


Figure 17.13

The Map interface maps keys to values.

The update methods include clear, put, putAll, and remove. The clear() method removes all mappings from this map. The put(Object key, Object value) method associates the specified value with the specified key in this map. If the map previously contained a mapping for this key, the old value associated with the key is returned. The putAll(Map m) method adds the specified map to this map. The remove(Object key) method removes the mapping for the specified key from this map.

The query methods include containsKey, containsValue, and isEmpty, and size. The containsKey(Object key) checks if the map contains a mapping for the specified key. The containsValue(Object value) method checks if the map contains a mapping for this value. The isEmpty() method checks if the map contains any mappings. The size() method returns the number of mappings in the map.

You can obtain a set of keys in the map using the keySet() method, and obtain a collection of values in the map using the values() method. The entrySet() method returns a collection of objects that implement Map.Entry interface. Each object in the collection is a specific key-value pair in the underlying map.

The AbstractMap class is a convenience class that implements all the methods in the Map interface, except the entrySet() method.

The SortedMap interface extends the Map interface to maintain the mapping in an ascending order of keys.

The HashMap and TreeMap classes are two concrete implementations of the Map interface. The HashMap class is efficient for locating a value, inserting a mapping, and deleting a mapping. The TreeMap class, implementing SortedMap, is efficient for traversing the keys in a sorted order. The keys can be sorted using the Comparable interface, or using the Comparator interface. If you create a TreeMap using its default constructor, the compareTo method in the Comparable interface is used to compare the elements in the set, assuming that the class of the elements implements the Comparable interface. To use a comparator, you have to use the constructor TreeMap(Comparator comparator) to create a sorted map that uses the compare method in the comparator to order the elements in the set.

Example 17.5 Using HashMap and TreeMap

This example creates a hash map that maps mortgages to borrowers. The Mortgage class, introduced in Chapter 5, "Programming with Objects and Classes," was used to model mortgages. Recall that you can create a mortgage using the following constructor

```
public Mortgage(double annualInterestRate, int numOfYear,  
double loanAmount)
```

The program first creates a hash map with the borrower's name as its key and mortgage as its value. The program then creates a tree map from the hash map, and displays the mappings in ascending order of the keys. The output of the program is shown in Figure 17.14.

```
package chapter17;  
  
import java.util.*;  
import chapter5.*;  
  
public class TestMap  
{  
public static void main(String[] args)  
{  
    // Create a hash set  
    HashMap hashMap = new HashMap();  
    hashMap.put("John F Smith", new Mortgage(7, 15, 150000));  
    hashMap.put("Greg Z Anderson", new Mortgage(7.5, 30, 150000));
```

```

HashMap.put("Joyce M Jones", new Mortgage(7, 15, 250000));
HashMap.put("Gerry K Lewis", new Mortgage(7.85, 30, 20000));
HashMap.put("Mathew T Cook", new Mortgage(7, 15, 100000));

// Display the loan amount for Gerry K Lewis
System.out.println("The loan amount for " + "Gerry K Lewis is " +
    ((Mortgage)(HashMap.get("Gerry K Lewis"))).getLoanAmount());

// Create a tree map from the previous hash map
TreeMap treeMap = new TreeMap(hashMap);

// Get an entry set for the tree map
Set entrySet = treeMap.entrySet();

// Get an iterator for the entry set
Iterator iterator = entrySet.iterator();

// Display mappings
System.out.println("\nDisplay mapping in ascending order of key");
while (iterator.hasNext())
{
    System.out.println(iterator.next());
}
}
}

```

```

C:\introjb4>java chapter17.TestMap
The loan amount for Gerry K Lewis is 20000.0

Display mapping in ascending order of key
Gerry K Lewis=chapter5.Mortgage@2f6684
Greg Z Anderson=chapter5.Mortgage@738798
John F Smith=chapter5.Mortgage@4b222f
Joyce M Jones=chapter5.Mortgage@3169f8
Mathew T Cook=chapter5.Mortgage@2457b6

C:\introjb4>

```

Figure 17.14

The program demonstrates using HashMap and TreeMap.

Example Review

All the concrete classes that implement the Map interface have at least two constructors. One is the default constructor that constructs an empty map, and the other constructor constructs a map from an instance of Map. So, new TreeMap(hashMap) constructs a tree map from a hash map.

Unlike the Collection interface, the Map interface does not provide an iterator. To traverse the map,

you create an entry set of the mappings using the `entrySet()` method in the `Map` interface.

TIP: If you don't need to maintain a sorted map when updating the map. You can use a hash map, because it is faster to insert and remove mappings in a hash map. When you need the map to be sorted, you can convert the map into a tree map.

The `Collections` Class

The `Collections` class contains various static methods for operating on collections and maps, for creating synchronized collection classes, and for creating read-only collection classes, as shown in Figure 17.15.

Collections
<code>+binarySearch(List list, Object key) : int</code>
<code>+binarySearch(List list, Object key, Comparator c) : int</code>
<code>+copy(List src, List des) : void</code>
<code>+enumeration(final Collection c) : Enumeration</code>
<code>+fill(List list, Object o) : void</code>
<code>+max(Collection c) : Object</code>
<code>+max(Collection c, Comparator c) : Object</code>
<code>+min(Collection c) : Object</code>
<code>+min(Collection c, Comparator c) : Object</code>
<code>+nCopies(int n, Object o) : List</code>
<code>+reverse(List list) : void</code>
<code>+reverseOrder() : Comparator</code>
<code>+shuffle(List list) : void</code>
<code>+shuffle(List list, Random rnd) : void</code>
<code>+singleton(Object o) : Set</code>
<code>+singletonList(Object o) : List</code>
<code>+singletonMap(Object key, Object value) : Map</code>
<code>+sort(List list) : void</code>
<code>+sort(List list, Comparator c) : void</code>
<code>+synchronizedCollection(Collection c) : Collection</code>
<code>+synchronizedList(List list) : List</code>
<code>+synchronizedMap(Map m) : Map</code>
<code>+synchronizedSet(Set s) : Set</code>
<code>+synchronizedSortedMap(SortedMap s) : SortedMap</code>
<code>+synchronizedSortedSet(SortedSet s) : SortedSet</code>
<code>+unmodifiedCollection(Collection c) : Collection</code>
<code>+unmodifiedList(List list) : List</code>
<code>+unmodifiedMap(Map m) : Map</code>
<code>+unmodifiedSet(Set s) : Set</code>
<code>+unmodifiedSortedMap(SortedMap s) : SortedMap</code>
<code>+unmodifiedSortedSet(SortedSet s) : SortedSet</code>

Figure 17.15

The `Collections` class contains static methods for supporting Java Collections Framework.

Most of the methods in the `Collections` class deal with lists. You can use the `sort` methods to sort a list using the `Comparable` interface or the `Comparator` interface. You can

use the binarySearch methods to find an element in a pre-sorted list. To use the binarySearch(list, key) method, the list must be previously sorted through the Comparable interface. To use the binarySearch(list, key, comparator) method, the list must be previously sorted through the Comparator interface.

You can use the copy(src, des) method to copy the source list to the destination list. You can use the fill(list, object) method to fill a list with the specified object. You can use the nCopy(n, object) method to create a list with n number of the specified object.

The min and max methods are generic for all the collections. You can use them to find the minimum and maximum element in a collection.

The Collections class defines three constants EMPTY_SET, EMPTY_LIST, and EMPTY_MAP for an empty set, an empty list, and an empty map. The class also provides the singleton(Object o) method for creating an immutable set containing only a single item, the singletonList(Object o) method for creating an immutable list containing only a single item, and the singletonMap(Object key, Object value) method for creating an immutable map containing only a single mapping.

The methods in the Collection and Map interfaces are not thread-safe. The Collections class provides six static methods synchronizedCollection(Collection c), synchronizedList(List list), synchronizedMap(Map m), synchronizedSet(Set set), synchronizedSortedMap(SortedMap m), and synchronizedSortedSet(SortedSet s) for wrapping a collection into a synchronized version. The synchronized collections can be safely accessed and modified by multiple threads concurrently.

The Collections class also provides six static methods unmodifiableCollection(Collection c), unmodifiableList(List list), unmodifiableMap(Map m), unmodifiableSet(Set set), unmodifiableSortedMap(SortedMap m), and unmodifiableSortedSet(SortedSet s) for creating read-only collections. The read-only collections not only prevent the data in the collections from being modified, but also offer better performance for read-only operations.

Example 17.6 Using the Collections Class

This example demonstrates using the methods in the Collections class. The example creates a list, sort it, and search an element. The example wraps the list

into a synchronized and read-only list. The output of the program is shown in Figure 17.16.

```
package chapter17;

import java.util.*;

public class TestCollections
{
    public static void main(String[] args)
    {
        // Create a list of three strings
        List list = Collections.nCopies(3, "red");

        // Create an array list
        ArrayList arrayList = new ArrayList(list);
        System.out.println("The initial list is " + arrayList);
        list = null; // Release list

        // Fill in "yellow" to the list
        Collections.fill(arrayList, "yellow");
        System.out.println("After filling yellow, the list is " +
            arrayList);

        // Add three new elements to the list
        arrayList.add("white");
        arrayList.add("black");
        arrayList.add("orange");
        System.out.println("After adding new elements, the list is\n"
            + arrayList);

        // Shuffle the list
        Collections.shuffle(arrayList);
        System.out.println("After shuffling, the list is\n"
            + arrayList);

        // Find the minimum and maximum elements in the list
        System.out.println("The minimum element in the list is "
            + Collections.min(arrayList));
        System.out.println("The maximum element in the list is "
            + Collections.max(arrayList));

        // Sort the list
        Collections.sort(arrayList);
        System.out.println("The sorted list is\n" + arrayList);

        // Find an element in the list
        System.out.println("The search result for gray is " +
            Collections.binarySearch(arrayList, "gray"));

        // Create a synchronized list
        List syncList = Collections.synchronizedList(arrayList);

        // Create a synchronized read-only list
        List unmodifiableList = Collections.unmodifiableList(syncList);
        arrayList = null; // Release arrayList
    }
}
```

```

    _____ syncList = null; // Release syncList

    _____ try
    _____ {
    _____     _____ unmodifiableList.add("black");
    _____ }
    _____ catch (Exception ex)
    _____ {
    _____     _____ System.out.println(ex);
    _____ }
    _____ }
}

```

```

C:\introjb4>java chapter17.TestCollections
The initial list is [red, red, red]
After filling yellow, the list is [yellow, yellow, yellow]
After adding new elements, the list is
[yellow, yellow, yellow, white, black, orange]
After shuffling, the list is
[white, black, orange, yellow, yellow, yellow]
The minimum element in the list is black
The maximum element in the list is yellow
The sorted list is
[black, orange, white, yellow, yellow, yellow]
The search result for gray is -2
java.lang.UnsupportedOperationException

C:\introjb4>

```

Figure 17.16

The program demonstrates using Collections.

Example Review

The program first creates a list filled with the same elements 3 times using `nCopies(3, "red")`. This list is an instance of `List`, but it is not an array list, nor a linked list. The program creates an array list from the list.

The program uses `Collections.fill(arrayList, "yellow")` to replace each element in the list with "yellow."

After adding three new elements into `arrayList`, the `Collections.shuffle(arrayList)` rearranges the elements in `arrayList`.

The program uses `Collections.min(arrayList)` to find the minimum element in `arrayList`, and uses

Collections.max(arrayList) to find the maximum element in arrayList.

Collections.sort(arrayList) is invoked to sort arrayList. Collections.binarySearch(arrayList, "gray") is invoked to find "gray" in arrayList. This method returns -1, if "gray" is not in arrayList.

The program finally uses Collections.synchronizedList(arrayList) to create a synchronized list for arrayList, and then creates a synchronized read-only list by wrapping the synchronized list using the unmodifiableList. As shown in Figure 17.16, an UnsupportedOperationException is thrown when the program attempts to add a new element to the read-only list.

The Arrays Class

The Arrays class contains various static methods for sorting and searching arrays, for comparing arrays, and for filling array elements. It also contains a method for converting an array to a list. Figure 17.17 shows in the methods in Arrays.

Arrays
<pre>+asList(Object[] a) : List +binarySearch(byte[] a, byte key) : int +binarySearch(char[] a, char key) : int +binarySearch(double[] a, double key) : int +binarySearch(float[] a, float key) : int +binarySearch(int[] a, int key) : int +binarySearch(long[] a, long key) : int +binarySearch(Object[] a, Object key) : int +binarySearch(Object[] a, Object key, Comparator c) : int +binarySearch(short[] a, short key) : int +equals(boolean[] a, boolean[] a2) : boolean +equals(byte[] a, byte[] a2) : boolean +equals(char[] a, char[] a2) : boolean +equals(double[] a, double[] a2) : boolean +equals(float[] a, float[] a2) : boolean +equals(int[] a, int[] a2) : boolean +equals(long[] a, long[] a2) : boolean +equals(Object[] a, Object[] a2) : boolean +equals(short[] a, short[] a2) : boolean +fill(boolean[] a, boolean val) : void +fill(boolean[] a, int fromIndex, int toIndex, boolean val) : void</pre>
Overloaded fill method for char, byte, short, int, long, float, double, and Object.
<pre>+sort(byte[] a) : void +sort(byte[] a, int fromIndex, int toIndex) : void</pre>
Overloaded sort method for char, short, int, long, float, double, and Object.

Figure 17.17

The Arrays class contains static methods for arrays.

The array must be sorted before the binarySearch method is used. The binarySearch method returns the index of the search key, if it is contained in the list. Otherwise, it return $-(\text{insertion point}) - 1$. The insertion point is the point at which the key would be inserted into the list. The fill method can be used to fill part of the array or the whole array with the same value. The sort method can be used to sort part of the array or the whole array.

Example 17.7 Using the Arrays Class

This example demonstrates using the methods in the Arrays class. The example creates an array of int values, fills part of the array with 50 elements, sorts it, searches for an element, and compares the array with another one, The output of the program is shown in Figure 17.18.

```
package chapter17;

import java.util.*;

public class TestArrays
{
    public static void main(String[] args)
    {
        // Create an array of 10 int values
        int[] array = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

        // Fill array from index 6 to index 8 with 50
        Arrays.fill(array, 6, 8, 50);
        System.out.println("After filling, the array is");
        for (int i=0; i<10; i++)
        {
            System.out.print(array[i] + " ");
        }
        System.out.println();

        // Sort the array
        Arrays.sort(array);
        System.out.println("After sorting, the array is");
        for (int i=0; i<10; i++)
        {
            System.out.print(array[i] + " ");
        }
        System.out.println();

        // Search for 30 in the array
        System.out.println("Search 30 in the array : " +
            Arrays.binarySearch(array, 30));
    }
}
```

```

    // Search for 3 in the array
    System.out.println("Search 3 in the array : " +
        Arrays.binarySearch(array, 3));

    // Search for -30 in the array
    System.out.println("Search -30 in the array : " +
        Arrays.binarySearch(array, -30));

    // Test if two arrays are the same
    int[] a = new int[10];
    System.out.println("Compare array with a : " +
        Arrays.equals(array, a));
}
}

```

```

C:\introjb4>java chapter17.TestArrays
After filling, the array is
0 1 2 3 4 5 50 50 8 9
After sorting, the array is
0 1 2 3 4 5 8 9 50 50
Search 30 in the array : -9
Search 3 in the array : 3
Search -30 in the array : -1
Compare array with a : false

C:\introjb4>

```

Figure 17.18

The program demonstrates using Arrays.

Example Review

The program first creates an array of 10 int values, fills 50 elements to array at index 6 and 7. The sort method is used to sort the entire array.

The program uses the binarySearch method to search for 30, 3, and -30 in the array. The return value is -9 for searching 30, because 30 is not in the list and the insertion point for 30 is at 8. The return value is 3 for searching 3, because 3 is in the list and its index is 3. The return value is -1 for searching -30, because -30 is not in the list and the insertion point for -30 is at 0.

The program also uses the equals method to compare two arrays.

Chapter Summary

This chapter introduced data structures using Java Collections Framework. You learned to use Set to store non-duplicate elements, use List to store sequenced elements, and use Map to store elements mapped with keys. Set, List and Map are interfaces.

HashSet and TreeSet are concrete implementation classes for Set. Elements in a HashSet are not ordered, but they are ordered in a TreeSet.

ArrayList and LinkedList are the concrete implementation classes for List. ArrayList uses a resizable array to store elements, and LinkedList uses a linked list to store elements. ArrayList is efficient for retrieving elements, and for inserting and removing elements from the end of the list. LinkedList is efficient for inserting and removing elements anywhere in the list. Vector is similar to ArrayList except that all methods in Vector are synchronized. Stack is a subclass of Vector that provides methods for stack operations.

HashMap and TreeMap are concrete implementation classes for Map. Use HashMap if the elements are not ordered, use TreeMap to order elements in a map.

The Collections class provides various static methods for operating on collections and maps, for creating synchronized collection classes, and for creating read-only collection classes.

The Arrays class contains various static methods for sorting and searching arrays, for comparing arrays, and for filling array elements.

Review Questions

- 7.1 Describe Java Collections Framework.
- 7.2 How do you create an instance of Set? How do you insert a new element in a set? How do you remove an element from a set? How do you find the size of a set? How do you traverse the elements in a set?
- 7.3 What are the differences between HashSet and TreeSet? How do you sort the elements in a set using the compareTo method in the Comparable interface? How do you sort the elements in a set using the Comparator interface?
- 7.4 How do you add, insert, and remove elements from a list? How do you traverse a list in both directions?
- 7.5 What are the differences between ArrayList and LinkedList?

- 7.6 How do you create an instance of Vector? How do you add or insert a new element into a vector? How do you remove an element from a vector? How do you find the size of a vector?
- 7.7 How do you create an instance of Stack? How do you add or insert a new element into a stack? How do you remove an element from a stack? How do you find the size of a stack?
- 7.8 How do you create an instance of Map? How do you add or insert a pair of element and key into a map? How do you remove an entry from a map? How do you find the size of a map? How do you traverse entries in a map?
- 7.9 Describe the static methods in the Collections class and Arrays class.

Programming Exercises

- 17.1 Create a program with two threads accessing and modifying a set concurrently. The first thread creates a hash set filled with numbers, and adds a new number to the set every one second. The second thread obtains an iterator for the set and traverses the set through the iterator back and forth every one second. You will receive a ConcurrentModificationException, because while traversing the set in the second thread, the underlying set is being modified in the first thread.
- 17.2 Correct the problem in the previous exercise using synchronization so the second thread does not throw ConcurrentModificationException.
- 17.3 Rewrite Example 17.4 using an ArrayList.
- 17.4 Read a text from a text file. Count the occurrences of the words using a hash map. Convert the hash map to a tree map. Display the words and their occurrences in ascending order of the words.
- (Hint: For each word, check if it is already a key in the map. If not, add the key and value 1 to the map. Otherwise, increase value for the word (key) by 1 in the map.)
- 17.5 Write a program to read in 10 double numbers and use the sort method in the Arrays class to sort these numbers.

