5.7 INTRODUCTION TO THE JAVA COLLECTION AND MAP INTERFACES (OPTIONAL SECTION)

Java provides several generic interfaces that are intended for collection classes that store Java objects. The two most basic of these interfaces are `Collection` and `Map`.

The Collection Interface

The methods of our own generic `Bag` are based on the generic `Collection<E>` interface, although we wanted to focus on basics, so we did not implement all of the `Collection` methods shown in Figure 5.6. Still, your experience with collection classes in Chapters 3 through 5 has prepared you for reading the documentation and using any of Java's classes that do implement the `Collection` interface. Some of the methods in that interface use a question mark as a data type, which is a feature that we'll discuss fully in Chapter 13.

The simplest Java `Collection` class is called `Vector`, which is similar to an array because you can place an object at a particular index. But unlike an array, a `Vector` automatically grows if you place an object at a location beyond its current capacity. Other classes that implement `Collection` include `Set`, `List`, `ArrayList`, `SortedSet`, and `HashSet`, some of which are discussed in Appendix D.

Vector
Set
List
ArrayList
SortedSet
HashSet

The Map Interface and the TreeMap Class

A *map class* is similar to a collection class in that a group of elements can be stored. However, the elements added to a map are called **keys**, and each key can have another object attached to it, called its **value**. The keys are usually small objects such as a social security number or a name. The values might be small bits of information, or they could be huge objects containing lots of information. The important concept about a map is that after a key/value pair is added to a map, the entire pair can be retrieved or removed simply by specifying its key. For example, we might build a map in which a student's ID number is the key and the value is an object that contains the student's entire academic record. When we need to retrieve a student's record, we can do so by specifying just the student's ID number.

Maps

A **map** is a collection class in which key/value pairs may be added. A pair can be retrieved or removed by specifying just its key.

Java has an interface called `Map`, and there are eight Java classes that implement this interface. The particular class that we will look at is the `TreeMap` class, which we discuss next.

FIGURE 5.6 Part of the API Documentation for the `Collection` Interface

From the Collection link in: http://java.sun.com/j2se/1.5.0/docs/api/index.html

java.lang

Interface Collection<E>

Method Summary

`boolean`	`add(E o)` Ensures that this collection contains the specified element.
`boolean`	`addAll(Collection<? extends E> c)` Adds all of the elements in the specified collection to this collection.
`void`	`clear( )` Removes all of the elements from this collection.
`boolean`	`contains(E o)` Returns true if this collection contains the specified element.
`boolean`	`containsAll(Collection<?> c)` Returns true if this collection contains all of the elements in the specified collection.
`boolean`	`equals(E o)` Compares the specified object with this collection for equality.
`int`	`hashCode( )` Returns the hash code value for this collection.
`boolean`	`isEmpty( )` Returns true if this collection contains no elements.
`Iterator<E>`	`iterator( )` Returns an iterator over the elements in this collection.
`boolean`	`remove(E o)` Removes a single instance of the specified element from this collection, if it is present.
`boolean`	`removeAll(Collection<?> c)` Removes all of this collection's elements that are also contained in the specified collection.
`boolean`	`retainAll(Collection<?> c)` Retains only the elements in this collection that are contained in the specified collection.
`int`	`size( )` Returns the number of elements in this collection.
`Object[ ]`	`toArray( )` Returns an array containing all of the elements in this collection.

The TreeMap Class

The generic `TreeMap` class has two generic type parameters: `K` for the keys and `V` for the type of the values. However, Java's `java.util.TreeMap` class (which implements the `Map` interface) has one extra requirement: The keys must come from a class that implements the `Comparable<K>` interface (from page 281). This allows a `TreeMap` to activate `k1.compareTo(k2)` to compare two keys `k1` and `k2`.

TreeMap from java.util

Java's `TreeMap<K,V>` implements the map interface in an efficient way in which the keys are required to come from a class (such as `String`) that implements the `Comparable` interface.

The most important `TreeMap` operations are specified in Figure 5.7.

To illustrate the `TreeMap` operations, we will write a program that uses a `TreeMap` to keep track of the number of times various words appear in a text file. We will use the words as the keys, and the value for each word will be an integer. For example, if the word "starship" appears in the text file 42 times, then the key/value pair of "starship"/42 will be stored in the map. With this in mind, we will write some examples of `TreeMap` operations using these three variables:

```
TreeMap<String, Integer> frequencyData;
String word;    // A word from our text file, to be used as a key
Integer count;  // The number of times that the word appeared
                // in our text file, to be used as a value in the TreeMap.
```

Notice that keys will be strings, which allows us to use a `TreeMap` since the Java `String` class does implement the `Comparable<String>` interface. Also the variable `count` is an `Integer` rather than a simple `int`. This is because the values in a map must be objects rather than primitive values.

FIGURE 5.7 Partial Specification for the `TreeMap` Class which Implements the `Map` Interface

Generic Class TreeMap<K,V>

❖ **public class Treemap from the package java.util**

A `TreeMap<K,V>` implements Java's `Map` interface for a collection of key/value pairs. The keys (of type `K`) in a `TreeMap` are required to implement the `Comparable<K>` interface so that, for any two keys `x` and `y`, the return value of `x.compareTo(y)` is an integer value that is:

— negative if `x` is less than `y`
— zero if `x` and `y` are equal
— positive if `x` is greater than `y`

(continued)

(FIGURE 5.7 continued)

Partial Specification (see the API documentation for complete specification)

♦ **Constructor for the TreeMap<K,V> (see the API documentation for more constructors)**
`public TreeMap( )`
Initialize a `TreeMap` with no keys and values.

♦ **clear**
`public void clear( )`
Remove all keys and values from this `TreeMap`.
Postcondition:
This `TreeMap` is now empty.

♦ **containsKey**
`public boolean containsKey(K key)`
Determine whether the `TreeMap` has a particular key.
Parameters:
`key` – the key to be searched for
Precondition:
The key can be compared to other keys in the `TreeMap` using the comparison operation.
Returns:
The return value is `true` if this `TreeMap` has a key of the specified value; otherwise, it's `false`.
Postcondition:
This `TreeMap` is now empty.
Throws: `ClassCastException` or `NullPointerException`
Indicates that the specified key cannot be compared to other keys currently in the `TreeMap`. (The `NullPointerException` means that the specified key is null, and the comparison operation does not permit null.)

♦ **get**
`public V get(K key)`
Gets the value that is currently associated to the specified key.
Parameters:
`key` – the key whose associated value is to be returned
Precondition:
The key can be compared to other keys in the `TreeMap` using the comparison operation.
Returns:
The value for the specified key within this `TreeMap`; if there is no such value, then the return value is null.
Throws: `ClassCastException` or `NullPointerException`
Indicates that the specified key cannot be compared to other keys currently in the `TreeMap`. (The `NullPointerException` means that the specified key is null, and the comparison operation does not permit null.)

(continued)

(FIGURE 5.7 continued)

♦ **keySet**
`public Set<K> keySet( )`
Obtain a `Set` that contains all the current keys of this `TreeMap`.
Returns:
The return value is a Java `Set` from the class `java.util.Set`. This `Set` is a container that contains all of the keys currently in this `TreeMap`.
Note:
Format for a loop that steps through every key in a `TreeMap t` (assuming the keys are strings):

```
String key;
while (K nextKey : t.keySet( ))
{
    ...process the next key, which is stored in nextKey...
}
```

♦ **put**
`public V put(K key, V value)`
Put a new key and its associated value into this `TreeMap`.
Parameters:
`key` and `value` – the key and its associated value to put into this `TreeMap`
Precondition:
The key can be compared to other keys in the `TreeMap` using the comparison operation.
Postcondition:
The specified key and its associated value have been inserted into this `TreeMap`. The return value is the value that was previously associated with the specified key (or null if there was no such key previously in the `TreeMap`).
Throws: `ClassCastException` or `NullPointerException`
Indicates that the specified key cannot be compared to other keys currently in the `TreeMap`. (The `NullPointerException` means that the specified key is null, and the comparison operation does not permit null.)
Note:
The return value does not need to be used. For example, `t.put(k,v)` can be a statement on its own.

♦ **size**
`public int size( )`
Obtain the number of key/value pairs currently in this `TreeMap`.
Returns:
The number of key/value pairs currently in this `TreeMap`.

Here are the common tasks we'll need to do with our `TreeMap`:

1. Putting a Key/Value Pair into a `TreeMap`. A key and its associated value are put into a `TreeMap` with the `put` method. For our example, we will read an English word into the variable `word` and compute the `count` of how many times the word occurs. Then we can put the `word` and its `count` into the `frequencyData` `TreeMap` with the statement:

```
frequencyData.put(word, count);
```

This adds a new key (`word`) with its value (`count`) to the `frequencyData`. If the `word` was already present in the `TreeMap`, then its old value is replaced by the new `count`.

2. Checking Whether a Specified Key Is Already in a TreeMap. The boolean method `containsKey` is used for this task. For example, the expression `frequencyData.containsKey(word)` will be true if the map already has a key that is equal to `word`.

3. Retrieving the Value That Is Associated with a Specified Key. The `get` method retrieves a value for a specified key. For example, the return value of `frequencyData.get(word)` is the `Integer` value associated with the key `word`. For our program, this return value is a Java `Integer` object.

4. Stepping Through All the Keys of a `TreeMap`. For any `TreeMap`, we can use the enhanced form of the for-loop to step through all the different keys currently in the map. The pattern for doing this uses the `keySet` method, as shown here for our word counting program:

```
for(String word : wordMap.keySet( ))
{
    ...do processing for this key, which is in the variable word...
```

This programming pattern works because the return value of `keySet` is a collection class that implements the `Iterable` interface.

The Word Counting Program

Using a `TreeMap` and the four operations we have just described, we can write a small program that counts the number of occurrences of every word in a text file. The program we write will just read the words (which are expected to be separated by spaces) and then print a table of this sort:

```
-----------------------------------------------------------
 Occurrences  Word
           2  aardvark
          10  dog
           1  not
           1  shower
-----------------------------------------------------------
```

In this example, the file contained four different words ("aardvark," "dog," "not," and "shower"). The word "aardvark" appeared twice, "dog" appeared 10 times, and the other two words appeared once each.

One of the key tasks in our program is to open the input file (which will be called `words.txt`) and read all the words in the file, compute the correct counts as we go, and store these counts in a `TreeMap` called `frequencyData`. The pseudocode for this task follows these steps:

```
A. Open the words.txt file for reading. We will use a Scanner object to do
   this (see Appendix B).

B. while there is still input in the file
   {
      word = the next word (read from the file)

      Get the current count (from frequencyData) of how many times the
      word has appeared in the file.

      Add one to that current count and store the result back in the
      count variable.

      frequencyData.put(word, count);
   }
```

The implementation of this pseudocode is given in the `readWordFile` method of Figure 5.8, along with the implementations of three other methods for the application. The `getCount` method is needed to get the current count (from `frequencyData`) of a word. In addition to getting the count, it converts from an `Integer` to an ordinary `int`. The `printAllCounts` method is particularly interesting because it uses an enhanced for-loop as discussed in Section 5.7.

Self-Test Exercises for Section 5.7

34. Write a Java statement that will put a key `k` into a `TreeMap` `t` with an associated value `v`. What will this statement do if `t` already has the key `k`?

35. Write an expression that will be true if a `TreeMap` `t` has a specific key `k`.

36. Suppose that a `TreeMap t` has a key `k` and that the value associated with `k` is an array of double numbers. Write a Java statement that will retrieve the value associated with `k` and assign it to an array variable called `v`.
37. Suppose `t` is a `TreeMap` with keys that are strings. Write a few Java statements that will print a list of all the keys in `t`, one per line.

FIGURE 5.8 Implementation of the `WordCounter` Program to Illustrate the Use of a `TreeMap`

Java Application Program

```
// File: WordCounter.java
// Program from Section 5.7 to illustrate the use of TreeMaps and Iterators.
// The program opens and reads a file called words.txt.
// Each line in this file should consist of one or more English words separated by spaces.
// The end of each line must not have any extra spaces after the last word.
// The program reads the file and then a table is printed of
// all words and their counts.

import java.util.*; // Provides TreeMap, Iterator and Scanner
import java.io.*;   // Provides FileReader and FileNotFoundException

public class WordCounter
{
   private static void main(String[ ] args)
   {
      TreeMap<String, Integer> frequencyData =
         new TreeMap<String, Integer>( );

      readWordFile(frequencyData);
      printAllCounts(frequencyData);
   }

   private static int getCount
   (String word, TreeMap<String, Integer> frequencyData)
   {
      if (frequencyData.containsKey(word))
      {  // The word has occurred before, so get its count from the map
         return frequencyData.get(word); // Auto-unboxed
      }
      else
      {  // No occurrences of this word
         return 0;
      }
   }
```

(continued)

(FIGURE 5.8 continued)

```
   private static void printAllCounts
   (TreeMap<String, Integer> frequencyData)
   {
      System.out.println("-----------------------------------------------");
      System.out.println("    Occurrences    Word");

      for(String word : frequencyData.keySet( ))
      {
         System.out.printf("%15d    %s\n", frequencyData.get(word), word);
      }

      System.out.println("-----------------------------------------------");
   }


   private static void readWordFile
   (TreeMap<String, Integer> frequencyData)
   {
      Scanner wordFile;
      String word;     // A word read from the file
      Integer count;   // The number of occurrences of the word

      try
      {  // Try to open the words.txt file:
         wordFile = new Scanner(new FileReader("words.txt"));
      }
      catch (FileNotFoundException e)
      {  // If the file failed, then print an error message and return without counting words:
         System.err.println(e);
         return;
      }

      while (wordFile.hasNext( ))
      {
         // Read the next word and get rid of the end-of-line marker if needed:
         word = wordFile.next( );

         // Get the current count of this word, add one, and then store the new count:
         count = getCount(word, frequencyData) + 1; // Autobox
         frequencyData.put(word, count);
      }
   }

}
```

CHAPTER SUMMARY

- A Java variable can be one of the eight primitive data types. Anythi that's not one of the eight primitive types is a reference to a Java Objec
- An assignment `x = y` is a **widening conversion** if the data type of x capable of referring to a wider variety of things than the type of y. It is **narrowing conversion** if the data type of x is capable of referring to smaller variety of things than the type of y. Java always permits wideni conversions, but narrowing conversions require a typecast.
- A **wrapper class** is a class in which each object holds a primitive valu Java provides wrapper classes for each of the eight primitive types. I many situations, Java will carry out automatic conversions from a primi tive value to a wrapper object (**autoboxing**) or the other way (**auto unboxing**).
- A **generic method** is similar to an ordinary method with one importna difference: The definition of a generic method can depend on an underly ing data type. The underlying data type is given a name, such as T, but T is not pinned down to a specific type anywhere in the method's implementa tion.
- When a class depends on an underlying data type, the class can be imple mented as a **generic class**. Converting a collection class to a generic class that holds objects is usually a small task. For example, we converted the `IntArrayBag` to an `ArrayBag` by following the steps on page 258.
- An interface provides a list of methods for a class to implement. By writ ing a class that implements one of the standard Java interfaces, you make it easier for other programmers to use your class. There may also be exist ing programs already written that work with some of the standard inter faces.
- Java's `Iterator<E>` generic interface provides an easy way to step through all the elements of a collection class. A class that implements Java's `Iterator` interface must provide two methods:
  ```
  public boolean hasNext( )
  public E next( )
  ```
  An `Iterator` must also have a `remove` method, although if removal is not supported, then the `remove` method can simply throw an exception.
- Two classes in this chapter have wide applicability, and you'll find them useful in the future: (1) the `Node` class from Appendix E, which is a node from a linked list of objects; and (2) the `LinkedBag` class from Appendix F, which includes a method to generate an `Iterator` for its elements.
- Java provides several different standard collection classes that implement the `Collection` interface (such as `Vector`) and the `Map` interface (such as `TreeMap`).

. The elements in the new bag are Java objects rather than integers. Also, the new bag has an `iterator` method to return a `Lister`.

). The bag from this section stores its elements on a linked list rather than in an array. Also, the new bag has an `iterator` method to return a `Lister`.

1. This uses three import statements:

```
import java.util.Scanner;
import
 edu.colorado.collections.LinkedBag;
import edu.colorado.nodes.Lister;
```

The code is:

```
Scanner stdin =
  new Scanner(System.in);
LinkedBag<String> b =
  new LinkedBag<String>( );
Lister<String> list;
String s;
int i;
for (i = 1; i <= 10; i++)
{
   System.out.print("Next: ");
   s = stdin.next( );
   b.add(s);
}
list = b.iterator( );
while (list.hasNext( ))
{
   s = list.next( );
   System.out.println(s);
}
```

32. An internal iterator is quick to implement and use, but an external iterator provides more flexibility, such as the ability to have two or more iterators active at once.

33. Any collection that implements the `Iterable` interface.

34. `t.put(k,v);` If `k` is already a key in `t`, then the `put` method will replace the old value with the new value `v`, and the return value of `put` will be the old value.

35. `t.containsKey(k)`

36. `v = (double [ ]) t.get(k);`

37.
```
Iterator it = t.keySet( ).iterator;
while (it.hasNext( ))
{
   System.out.println(it.next( ));
}
```

PROGRAMMING PROJECTS

1 Implement a generic class for a sequence of Java objects. You can store the objects in an array (as in Section 3.3) or in a linked list (as in Section 4.5). The class should also implement the `Iterable` interface.

2 Write a program that uses a bag of strings to keep track of a list of chores you have to accomplish today. The user of the program can request several services: (1) Add an item to the list of chores; (2) Ask how many chores are in the list; (3) Print the list of chores to the screen; (4) Delete an item from the list; (5) Exit the program.

If you know how to read and write strings from a file, then have the program obtain its initial list of chores from a file. When the program ends, it should write all unfinished chores back to this file.

3 For this project, you will use the bag class from Appendix F, including the `grab` method that returns a randomly selected element. Use this ADT in a program that does the following: