Listing 8-10: *Modifying the* `Alarm` *class to play a sound* (continued)

```
22          System.exit(1);
23        }
24      }
25    }
26
27    public void ring()
28    { // Same as Listing 8-6.
29
30      // Play the sound.
31      Alarm.sound.play();
32    }
33  }
```

8.5 Java's Collection Classes

Programs often need to have collections of similar objects. The alarm clock program we developed in the previous section is a prime example. Even with a collection of only four alarms, code such as `setAlarm` and `checkAndRingAlarms` got tedious. Furthermore, why should there be only four alarms? Why not 40 or 400 or even 4 million?

Four million alarms seems excessive, but other programs could easily have a collection of 4 million or more objects. Consider an inventory program for a large chain of stores, for example. When our collections of similar objects grow beyond four or five, we need better techniques than we used in `AlarmClock`.

Fortunately, Java provides a set of classes for maintaining collections of objects. These classes are used when objects in a collection need to be treated in a similar way: a collection of `Alarm` objects that need to be checked and perhaps rung, a collection of `Student` objects that need to be enrolled in a course, or a collection of `Image` objects that need to display on a computer monitor. The objects maintained by these collections are usually called the **elements** of the collection.

Java has three kinds of collections:

- A **list** is an ordered collection of elements, perhaps with duplicates. Because the list is ordered, you can ask for the element in position 5, for example.
- A **set** is an unordered collection of unique elements; duplicates are not allowed.
- A **map** is an unordered collection of associated **keys** and **values**. A key is used to find the associated value in the collection. For example, your student number is a key that is often used to look up an associated value, such as your address or grades.

Collection objects cannot hold primitive types, only objects. We'll discuss a way around that limitation in Section 8.5.4.

KEY IDEA

Collections hold objects, not primitives.

These collection classes are sophisticated, and covering all the details would require several chapters. Therefore, we will focus on constructing the objects; adding and removing elements, plus a few other useful methods; and processing all the elements (for example, checking all the `Alarm` objects to see if one should be rung). We'll look at one example of each kind of collection. We'll look at a list class first in some detail. We will go faster when we examine sets and maps because much of what we learn with lists will also apply to them.

The approach taken in this textbook assumes that you are using Java 5.0 or higher. Previous versions of Java have these classes, but they are more difficult to use without the advances made in Java 5.0

KEY IDEA

This section assumes you are using Java 5.0 or higher.

8.5.1 A List Class: `ArrayList`

A list is probably the most natural collection class to use for our `AlarmClock` program. It can hold any kind of object (sets and maps have some restrictions) and allows us to easily process all of the elements or to get just one.

There are two distinct ways to write a list class—`ArrayList` and `LinkedList`. Both are in the `java.util` package, meaning that you'll need to import from that package if you want to use the classes. `ArrayList` is the one we'll study here. By the end of Chapter 10, you will be able to write a simple version of `ArrayList`. By the end of your second computer science course, you should be able to write your own version of `LinkedList`.

Lists such as `ArrayList` keep its elements in order. It makes sense to speak of the first element or the last element. Like a `String`, an individual element is identified by its index—a number greater than or equal to zero and less than the number of elements in the list. The number of elements in the list can be obtained with the `size` query.

KEY IDEA

The `size` query returns the number of elements in the list.

Construction

The type of a collection specifies the collection's class and the class of object it holds. For example, one type that could hold a collection of `Alarm` objects is `ArrayList<Alarm>`. The type of objects held in the collection is placed between angle brackets. This type can be used to declare and initialize a variable, as follows:

```
ArrayList<Alarm> alarms = new ArrayList<Alarm>();
```

KEY IDEA

The type of the collection includes the type of objects to be stored in it.

A list of `Robot` objects and a list of `Person` objects would be created similarly:

```
ArrayList<Robot> workers = new ArrayList<Robot>();
ArrayList<Person> friends = new ArrayList<Person>();
```

Of course, if we're declaring instance variables, we would include the keyword `private` at the beginning of each line.

KEY IDEA

A collection allows you to access many objects using only one variable.

In the `AlarmClock` class shown in Listing 8-7, the declaration of the four `Alarm` instance variables in lines 10–13 can be replaced with the following line:

```
private ArrayList<Alarm> alarms = new ArrayList<Alarm>();
```

Furthermore, we are no longer limited to just four alarms.

FIND THE CODE

ch08/alarmsWithLists/

Adding Elements

The power of using a collection class becomes evident in the `setAlarm` method. In Listing 8-7, we devote lines 68–78 to assigning an alarm to one of the four instance variables—11 lines. Even so, we're limited to only four alarms. For each additional alarm, we need to add an instance variable and two more lines in the `setAlarm` method.

Using an `ArrayList` to store the alarms reduces lines 68–78 to a single line:

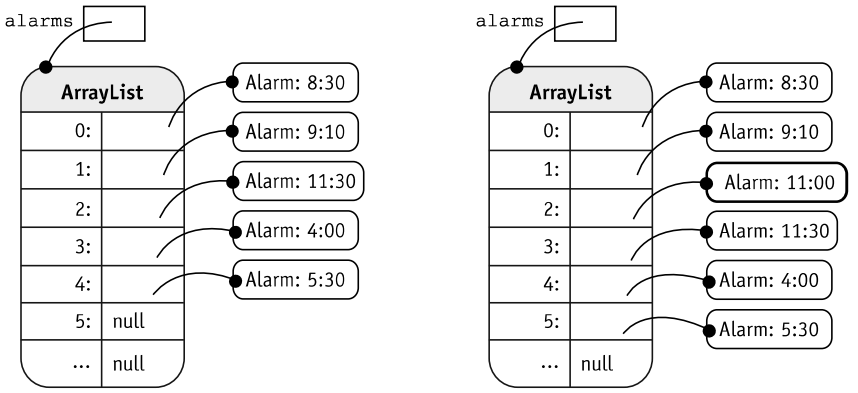
```
this.alarms.add(theAlarm);
```

Furthermore, we can now have an almost unlimited number of alarms.

The `add` method just shown adds the new alarm to the end of the list. An overloaded version of `add` allows you to state the index in the list where the alarm should be added. Like `Strings`, an `ArrayList` numbers the positions in its list starting with 0. Therefore, the following line adds a new alarm in the third position:

```
this.alarms.add(2, theAlarm);
```

The alarms at indices 0 and 1 come before it. Objects at indices 2 and larger are moved over by one position to make room for the new object. Figure 8-18 illustrates inserting a new `Alarm` for 11:00 at index 2.



Before inserting an `Alarm` at `index 2`

After inserting an `Alarm` at `index 2`

(figure 8-18)

Inserting an `Alarm` *into an* `ArrayList` *at index 2*

The index for `add` must be in the range `0..size()`. Positions can't be skipped when adding objects. For example, you can't add an object at index 2 before there is data at indices 0 and 1. Doing so results in an `IndexOutOfBoundsException`.

Getting, Setting, and Removing Elements

A single element of the collection can be accessed using the `get` method and specifying the object's index. For example, to get a reference to the third alarm (which is at index 2 because numbering starts at 0), write the following statements:

```
Alarm anAlarm = this.alarms.get(2);
anAlarm.ring();              // do something with the alarm
```

As with any other method that returns a reference, you aren't required to assign the reference to a variable before calling a method. We could condense the previous two statements to a single line:

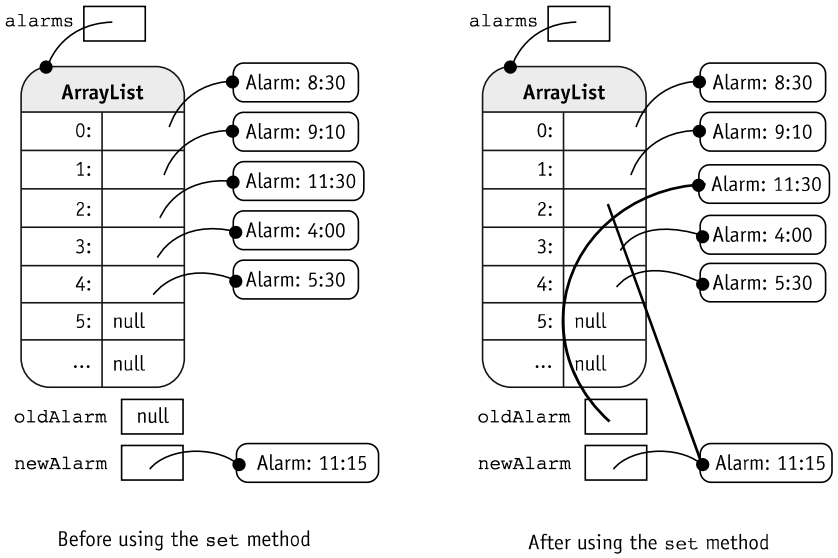
```
this.alarms.get(2).ring();
```

An element can be replaced using the `set` method. Its parameters are the index of the element to replace and the object to put there. For example, Figure 8-19 illustrates the change made by the following code fragment:

```
Alarm oldAlarm = null;
Alarm newAlarm = new Alarm(11, 15, "Meeting with Mohamed");
oldAlarm = this.alarms.set(2, newAlarm);
```

(figure 8-19)

Effects of the `set` *method*



Notice that the element at index 2 now refers to the new alarm. The `set` method returns a reference to the element that is replaced, which is assigned to `oldAlarm`.

An element can be removed from the `ArrayList` with the `remove` method. Its only argument is the index of the element to remove. After removing the element, any elements in subsequent positions are moved up to occupy the now open position—the opposite of what `add` does. Like `set`, `remove` returns a reference to the removed element.

Other Useful Methods

There are many other methods in the `ArrayList` class and its superclasses. Table 8-1 lists the name and purpose of some of the most useful methods. `E` represents the type of elements stored in this particular collection.

The `contains` and `indexOf` methods depend on the element's class overriding the equals method to test for equivalence. As noted in Section 8.2.4, we don't have the tools to do this yet for the classes we write. Provided classes such as `String`, `DateTime`, and others should meet this requirement.

Method	Purpose
`boolean add(E elem)`	Add the specified element to the end of this list. Return `true`.
`void add(int index, E elem)`	Insert the specified element at the specified index in this list. `0 ≤ index < size()`.
`void clear()`	Remove all of the elements from this list.
`boolean contains(Object elem)`	Return `true` if this list contains the specified element.
`E get(int index)`	Return the element at the specified index. `0 ≤ index < size()`.
`int indexOf(Object elem)`	Search for the first element in this list that is equal to `elem`, and return its index or `-1` if there is no such element in this list.
`boolean isEmpty()`	Return `true` if this list contains no elements.
`E remove(int index)`	Remove and return the element at the given index. `0 ≤ index < size()`.
`E set(int index, E elem)`	Replace the element at the given position in this list with `elem`. Return the old element. `0 ≤ index < size()`.
`int size()`	Return the number of elements in this list.

(table 8-1)

Some of the most useful methods in the `ArrayList` *class.* `E` *is the type of the elements*

Processing All Elements

The last detail needed to replace the four `Alarm` variables with a list is checking each alarm to see if it's time to ring it. In Listing 8-7, we did this in lines 47–52. Each line calls a helper method to check one of the alarms. That means 4 alarms, 4 lines of code; 400 alarms, 400 lines of code.

There are three distinct ways[3] to process all of the elements in an `ArrayList`. We've already seen the basic tools for one of them: the `get` and `size` methods. We can use them in a `for` loop to get each element in turn:

```
1 private void checkAndRingAlarms(DateTime currTime)
2 { for (int index = 0; index < this.alarms.size(); index++)
3   { Alarm anAlarm = this.alarms.get(index);
4     this.checkOneAlarm(anAlarm, currTime);
5   }
6 }
```

Process All Elements

[3] The third way uses iterators, a topic we won't be covering in this textbook.

These six lines of code completely replace `checkAndRingAlarms` in lines 47–52 of Listing 8-7. Furthermore, this code will work for almost[4] any number of alarms—from zero on up.

A loop to process all of the elements in a collection is so common that Java 5.0 introduced a special version of the `for` loop just to make these situations easier. It is sometimes called a **foreach** loop—the body of the loop executes once for each element in the collection.

Using a *foreach* loop to process each alarm results in the following method:

Process All Elements

```
private void checkAndRingAlarms(DateTime currTime)
{ for(Alarm anAlarm : this.alarms)
  { this.checkOneAlarm(anAlarm, currTime);
  }
}
```

A template for the *foreach* loop is as follows:

```
for(«elementType» «varName» : «collection»)
{«statements using varName»
}
```

The statement includes the keyword `for`, but instead of specifying a loop index, the `for each` loop declares a variable, *«varName»*, of the same type as the objects contained in *«collection»*. The variable name is followed with a colon and the collection that we want to process. *«varName»* can only be used within the body of the loop.

A version of `AlarmClock` that uses an `ArrayList` is shown in Listing 8-11. Note that changes are shown in bold. Documentation is identical to Listing 8-7, so it is omitted.

FIND THE CODE

cho8/alarmsWithLists/

Listing 8-11: *The* `AlarmClock` *class implemented with an* `ArrayList`

```
1  import becker.util.DateTime;
2  import becker.util.Utilities;
3  import java.util.ArrayList;
4
5  public class AlarmClock extends Object
6  {
7    // A list of alarms.
8    private ArrayList<Alarm> alarms = new ArrayList<Alarm>();
9
10   private int numAlarmsLeft = 0;
```

[4] We don't say `ArrayList` will handle any number because eventually your computer would run out of memory to store them all.

Listing 8-11: *The* `AlarmClock` *class implemented with an* `ArrayList` (continued)

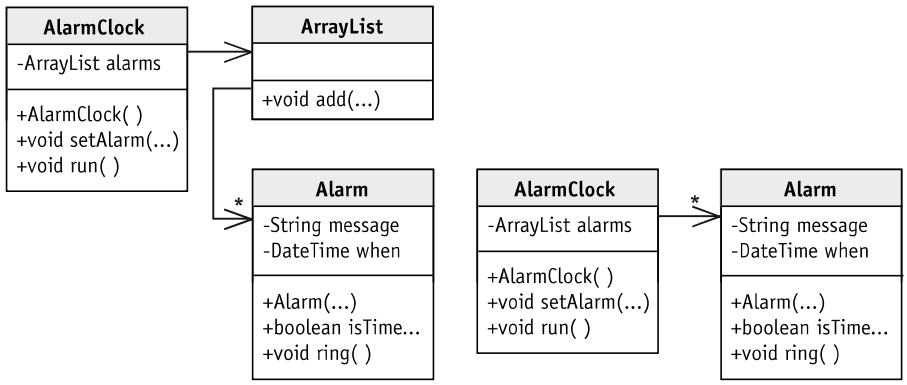
```
11    private final boolean TESTING;
12
13    public AlarmClock(boolean test)
14    { // Same as Listing 8-7.
15    }
16
17    public void run(int secPerSec)
18    { // Same as Listing 8-7.
19    }
20
21    private void checkAndRingAlarms(DateTime currTime)
22    { for(Alarm anAlarm : this.alarms)
23      { this.checkOneAlarm(anAlarm, currTime);
24      }
25    }
26
27    private void checkOneAlarm(Alarm alarm, DateTime currTime)
28    { // Same as Listing 8-7.
29    }
30
31    public void setAlarm(int hr, int min, String msg)
32    { Alarm theAlarm = new Alarm(hr, min, msg);
33      this.alarms.add(theAlarm);
34      this.numAlarmsLeft++;
35    }
36
37    public static void main(String[] args)
38    { // Same as Listing 8-7.
39    }
40  }
```

Class Diagrams

Someone drawing a class diagram for `AlarmClock`, as shown in Listing 8-11, would probably draw a diagram as shown in Figure 8-20a. However, collection classes like `ArrayList` appear so often in Java programs and their function is so well known that most programmers prefer to draw the abbreviated class diagram shown in Figure 8-20b.

(figure 8-20)

Class diagrams



a. Full class diagram

b. Abbreviated class diagram

8.5.2 A Set Class: `HashSet`

Like a list, a set also manages a collection of objects. There are two important differences:

- A set does not allow duplicate elements. Sets ignore attempts to add an element that is already in the set.
- The elements are not ordered. None of the methods in `HashSet` take an index as an argument.

KEY IDEA

Sets do not allow duplicates.

These restrictions don't affect the `AlarmClock` class—each alarm is unique and individual alarms are not important; they are all processed as a group. In fact, changing `ArrayList` to `HashSet` in line 8 of Listing 8-11 is all that is needed to convert that program to use a set.

LOOKING AHEAD

Processing files is a major topic of Chapter 9.

So how might we exploit the specific properties of a set? We could use it, for example, to count the number of unique strings in a file. About two dozen lines of code are enough to discover that William Shakespeare's play *Hamlet* contains 7,467 unique "words." (Words is quoted because the program doesn't remove punctuation or numbers, meaning that "merry" and "merry?" are considered different words.)

Construction

We'll use an instance of the `HashSet` class to count the words. An instance of `HashSet` is constructed just like `ArrayList`—specify the type of the elements you want it to manage in angle brackets. In this case, we'll store our words as strings.

```
HashSet<String> words = new HashSet<String>();
```

Useful Methods

Words can be added to this set with the `add` method. If the word is already there, it will be ignored.

To add many words, we should read them from a file—the topic of Section 9.1. Until then, we can add some words from *Hamlet* manually:

```
words.add("to");
words.add("be");
words.add("or");
words.add("not");
words.add("to");
words.add("be");
```

FIND THE CODE

cho8/collections/

The `size` method returns the number of elements in the set. Given the previous six calls to `add`, `size` would return 4.

The word "not" could be removed with the statement `words.remove("not")`. In general, an object is removed from the set by passing the object to the `remove` method.

The `contains` method will return `true` if the set contains the given object and `false` otherwise. Other useful methods are summarized in Table 8-2.

(table 8-2)

Some of the most useful methods in the `HashSet` *class (*`E` *is the type of the elements)*

Method	Purpose
`boolean add(E elem)`	Add the specified element to this set. Return `true` if the element was already present.
`void clear()`	Remove all of the elements from this set.
`boolean contains(Object elem)`	Return `true` if this set contains the specified element.
`boolean isEmpty()`	Return `true` if this set contains no elements.
`boolean remove(Object elem)`	Remove the specified element from this set, if present. Return `true` if the element was present.
`int size()`	Return the number of elements in this set.

Processing All Elements

We can print all of the words in the set using a `for each` loop, just as we processed all of the elements in the `ArrayList` earlier.

KEY IDEA

A set's `for each` *loop works the same way as for a list.*

```
for (String w : words)
{ System.out.print(w + "  ");
}
```

PATTERN

Process All Elements

Executing this loop after adding the first six words of Hamlet's speech would yield "to," "be," "or," and "not." The order in which they are printed is *not* specified.

Limitations

`HashSet` uses a technique known as **hashing**, in which elements are stored in an order defined by the element's `hashCode` method. The hash code is carefully constructed to make operations such as `contains` and `remove` faster than for an `ArrayList`. When the elements are printed, however, they appear in an order that seems random.

`hashCode` is inherited from the `Object` class. As defined there, no two objects are considered equal or equivalent. If two elements in your set should be considered equivalent (for example, two different date objects both representing the same date), the `equals` and `hashCode` methods must both be overridden. Unfortunately, overriding `hashCode` is beyond the scope of this textbook. However, you should have no problem using `HashSet` if you either use it with a set of unique objects or use it with provided classes, such as `String` or `DateTime`.

8.5.3 A Map Class: `TreeMap`

KEY IDEA

A map associates a key with a value. Use the key to look up the value.

A map is a collection of associated keys and values. A key is used to find the associated value in the collection. For example, we could associate the names of our friends (the keys) with their phone numbers (the values), as shown in Figure 8-21.

(figure 8-21)

Key-value pairs

Key	Value
Sue	578-3948
Fazila	886-4957
Jo	1-604-329-1023
Don	578-3948
Rama	886-9521

With these associations between keys and values, we can ask questions such as "What's the phone number for Don?" We use the key, "Don," to look up the associated value, "578-3948."

KEY IDEA

The keys in any given map must be unique.

Notice that all the keys are unique; that's a fundamental requirement of a map. If we have two friends named "Don" we must distinguish between them, perhaps by adding initials or last names. However, the associated values do not need to be unique. In this example, Don and Sue both appear in the mapping even though they have the same phone number.

Java provides two classes implementing a map, `TreeMap` and `HashMap`. Each one has different advantages and disadvantages. `HashMap`s have the advantage of being somewhat faster but require a correct implementation of the `hashCode` method. On the other hand, `TreeMap`s keep the keys in sorted order but require a way to order the elements. We'll use a `TreeMap` to build a simple phone book.

Construction

When declaring and constructing a `TreeMap` object, the types for both the keys and the values must be specified. For our simple phone book, we'll use `String`s for both the keys and the values:

```
TreeMap<String, String> phoneBook =
                                new TreeMap<String, String>();
```

Although this example happens to use strings for both keys and values, that need not be the case. The types of the keys and values are often different and must be a reference type—not a primitive type like `int`, `double`, or `char`.

How can you figure out that `TreeMap` needs two types to define it but that `ArrayList` and `HashSet` require only one? Look at the class documentation. Figure 8-22 shows the beginning of the online documentation for `TreeMap`, which includes `TreeMap<K, V>` in large type. The two capital letters between the angle brackets indicate that two types are needed when a `TreeMap` is constructed. Finding out that `K` stands for the type of the key and `V` stands for the type of the value is, unfortunately, not as easy to figure out from the documentation.

There is one restriction on the type of the key. Because `TreeMap` keeps the keys in sorted order, it needs a way to compare them. It relies on the key's class to implement the `Comparable` interface. The keys are then known to have a `compareTo` method. `String` and `DateTime` both implement the interface and can be used as keys.

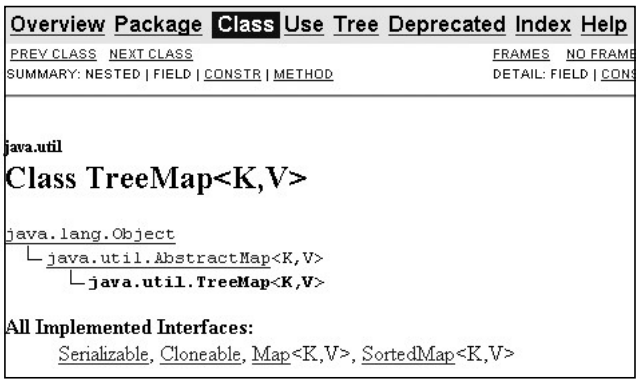
You can tell if a class implements `Comparable` by looking at the "All Implemented Interfaces" line in the documentation. You can see an example of this line in Figure 8-22. Also, if you look at the documentation for `Comparable`, it will list the classes in the Java library that implement it.

LOOKING AHEAD

Writing your own classes that implement the `Comparable` *interface will be discussed in Section 12.5.1.*

(figure 8-22)

Part of the online documentation for `TreeMap`



Useful Methods

Pairs are added to a map with the `put` method. It takes a key and a value as arguments:

```
phoneBook.put("Sue", "578-3948");
phoneBook.put("Fazila", "886-4957");
```

If the key already exists in the map, the value associated with that key will be replaced by the new value.

A value can be retrieved with the `get` method. The key of the desired value is passed as an argument. For example, after executing the following line:

```
String number = phoneBook.get("Sue");
```

the variable `number` will contain "578-3948" (assuming the associations shown in Figure 8-21). It's similar to accessing an element in a list except that instead of specifying the element's index, you specify the element's key.

The `remove` method takes a key as its only argument and removes both the key and its associated value.

Like a list and a set, a map has `isEmpty`, `clear`, and `size` methods. Instead of `contains`, it has two methods: `containsKey` and `containsValue`, which both return a Boolean result. These methods are summarized in Table 8-3.

Method	Purpose
`void clear()`	Remove all of the key-value pairs from this mapping.
`boolean containsKey (Object elem)`	Return `true` if this mapping contains the specified key.
`boolean containsValue (Object elem)`	Return `true` if this mapping contains the specified value.
`V get(Object key)`	Return the value associated with the specified key.
`boolean isEmpty()`	Return `true` if this mapping contains no elements.
`Set<K> keySet()`	Return a set containing the keys in this mapping.
`V put(K key, V value)`	Associate the specified key with the specified value in this mapping. Return the value previously associated with the key or `null` if there wasn't one.
`V remove(Object key)`	Remove and return the value associated with the specified key, if it exists. Return `null` if there was no mapping for the key.
`int size()`	Return the number of key-value pairs in this mapping.

(table 8-3)

Some of the most useful methods in the `TreeMap` *class (*`K` *is the type of the keys;* `V` *is the type of the values)*

Processing All Elements

Processing all the elements in a map is more complicated than a list or a set because each element is a pair of objects rather than just one thing.

One approach is to use the `keySet` method to get all of the keys in the map as a set. We can then loop through all of the keys using the `for each` loop. As part of the processing, we can also get the associated value, as shown in the following example:

```
// print the phoneBook
for (String key : phoneBook.keySet())
{ System.out.println(key + "=" + phoneBook.get(key));
}
```

Process All Elements

Completed Program

The completed telephone book program is shown in Listing 8-12. It uses a `Scanner` object in 27 and 30 to obtain a name from the program's user. Using `Scanner` effectively is one of the primary topics of the next chapter.

FIND THE CODE

cho8/collections/

Listing 8-12: *An electronic telephone book*

```
1  import java.util.*;
2
3  /** An electronic  telephone book.
4  *
5  *  @author Byron Weber Becker */
6  public class MapExample extends Object
7  {
8    public static void main(String[] args)
9    { // Create the mapping between names and phone numbers.
10     TreeMap<String, String> phoneBook =
11                               new TreeMap<String, String>();
12
13     // Insert the phone numbers.
14     phoneBook.put("Sue", "578-3948");
15     phoneBook.put("Fazila", "886-4957");
16     phoneBook.put("Jo", "1-604-329-1023");
17     phoneBook.put("Don", "578-3948");
18     phoneBook.put("Rama", "886-9521");
19
20     // Print the phonebook.
21     for (String k : phoneBook.keySet())
22     { System.out.println(k + "=" + phoneBook.get(k));
23     }
24
25     // Repeatedly ask the user for a name until "done" is entered.
26     // Scanner is discussed in detail in Chapter 9.
27     Scanner in = new Scanner(System.in);
28     while (true)
29     { System.out.print("Enter a name or 'done': ");
30       String name = in.next();
31
32       if (name.equalsIgnoreCase("done"))
33       { break;   // Break out of the loop.
34       }
35
36       System.out.println(name + ":" + phoneBook.get(name));
37     }
38   }
39 }
```

8.5.4 Wrapper Classes

What if we want to store integers or characters or some other primitive type in one of the collection classes? For example, we might need a set of the prime numbers (integers that can only be divided evenly by 1 and itself). If we write

```
HashSet<int> primeNumbers = new HashSet<int>();
```

the Java compiler will give us a compile-time error, perhaps with the cryptic message "unexpected type." The problem is that the compiler is expecting a reference type—the name of a class—between the angle brackets. `int`, of course, is a primitive type.

We can get around this by using a **wrapper class**. It "wraps" a primitive value in a class. A simplified wrapper class for `int` is as follows:

```
public class IntWrapper extends Object
{ private int value;

  public IntWrapper(int aValue)
  { super();
    this.value = aValue;
  }

  public int intValue()
  { return this.value;
  }
}
```

Fortunately, Java provides a wrapper class for each of the primitive types: `Integer`, `Double`, `Boolean`, `Character`, and so on. These are in the `java.lang` package, which is automatically imported into every class.

We can use these built-in wrapper classes to construct a set of integers:

```
HashSet<Integer> primes = new HashSet<Integer>();
```

KEY IDEA

Java 5.0 automatically converts between primitive values and wrapper classes.

The Java compiler will automatically convert between an `int` and an instance of `Integer` when using `primes`. For example, consider the program in Listing 8-13. In lines 12–17, the `add` method takes an `int`, not an instance of `Integer`. The `contains` method in line 25 is the same. Before Java 5.0 the programmer needed to manually include code to convert between primitives and wrapper objects.

FIND THE CODE

cho8/collections/

Listing 8-13: *A program to help classify prime numbers*

```
1  import java.util.*;
2
3  /** Help the user find out if a number is prime.
4  *
5  *  @author Byron Weber Becker */
6  public class WrapperExample extends Object
7  {
8    public static void main(String[] args)
9    { HashSet<Integer> primes = new HashSet<Integer>();
10
11     // The prime numbers we know.
12     primes.add(2);
13     primes.add(3);
14     primes.add(5);
15     primes.add(7);
16     primes.add(11);
17     primes.add(13);
18
19     // Help the user classify numbers.
20     // Scanner is discussed in detail in Chapter 9.
21     Scanner in = new Scanner(System.in);
22     System.out.print("Enter a number: ");
23     int num = in.nextInt();
24
25     if (primes.contains(num))
26     { System.out.println(num + " is prime.");
27     } else if (num <= 13)
28     { System.out.println(num + " is not prime.");
29     } else
30     { System.out.println(
31             num + " might be prime; it's too big for me to know.");
32     }
33   }
34 }
```

8.6 GUIs and Collaborating Classes

Programs with graphical user interfaces almost always use collaborating classes in two ways. Collaborating classes makes these programs easier to understand, write, debug, and maintain.