CHAPTER 14

INTRODUCTION TO COLLECTIONS

OBJECTIVES

After completing this chapter, you will be able to:

- Recognize different categories of collections and the operations on them
- Distinguish between a collection's interface and its implementing classes
- Use list, stack, queue, set, and map collections to solve problems
- Choose a particular collection implementation based on its performance characteristics

Estimated Time: 5 hours

VOCABULARY

Association list
Collection
Dictionary
Hashing
Iterator
Keyed list
List
Map
Queue
Set
Stack
Table
Type parameter
Type variable
Wrapper class

Thus far in this book, you have learned the basic elements of programming and problem solving. You have also learned how to organize a software system in terms of cooperating classes. In this chapter, we explore several frequently used classes called collections. Although they differ in structure and use, collections all have the same fundamental purpose—they help programmers to effectively organize data in programs.

Collections can be viewed from two perspectives. Users or clients of collections are concerned with the operations they provide. Developers or implementers of collections are concerned with coding these operations efficiently. In this chapter, we give an overview of different types of collections from the perspective of the users of these collections.

14.1 Overview of Collections

A *collection*, as the name implies, contains a group of items that we want to treat as a conceptual unit. Nearly every nontrivial piece of software involves the use of collections. For example, the strings and arrays discussed in earlier chapters are collections of characters and arbitrary

elements, respectively. Other important types of collections include lists, stacks, queues, sets, maps, trees, and graphs. Collections can be homogeneous, meaning that all items in the collection must be of the same type, or heterogeneous, meaning the items can be of different types. An important distinguishing characteristic of collections is the manner in which they are organized. Table 14-1 lists the most commonly used types of collections and gives some examples of their applications.

TABLE 14-1
Types of collections

TYPE OF COLLECTION	ORGANIZATION OF ELEMENTS	SAMPLE USE
String	Linear sequence of characters accessed by numeric index position	A line of text
List	Linear sequence of elements of any type by numeric index position	A shopping list or a list of instructions
Stack	A linear sequence of elements with access to end, called the top	A stack of dinner plates or a deck of cards
Queue	A linear sequence of elements with insertions at one end, called the rear, and removals from the other end, called the front	A line of customers in a supermarket checkout
Set	An unordered collection of unique elements	The persons in the president's cabinet
Map	An unordered collection of elements accessed by unique keys	A dictionary
Tree	A hierarchical collection of elements	An organizational chart
Graph	A network of elements	Airline flight paths between cities

In this chapter, we cover the use of all these collections except for trees and graphs, which are the subjects of more advanced computer science courses.

Operations on Collections

Collections are typically dynamic rather than static, meaning they can grow or shrink with the needs of a problem. Also, their contents normally can change throughout the course of a program. The manipulations that can be performed on a collection vary with the type of collection being used, but generally, the operations fall into several broad categories that are outlined in Table 14-2.

TABLE 14-2
Categories of operations on collections

CATEGORY OF OPERATION	DESCRIPTION
Search and retrieval	These operations search a collection for a given target item or for an item at a given position. If the item is found, either it or its position is returned. If the item is not found, a distinguishing value, such as `null` or –1, is returned.
Removal	This operation deletes a given item or the item at a given position.
Insertion	This operation adds an item to a collection, usually at a particular position within the collection.
Replacement	This operation combines removal and insertion into a single operation.
Traversal	This operation visits each item in a collection. Depending on the type of collection, the order in which the items are visited can vary. During a traversal, items can be accessed or modified. Collections that can be traversed with Java's enhanced `for` loop are said to be *iterable*.
Determine the size	This operation determines the size of a collection—the number of items it contains. Some collections also have a maximum capacity, or number of places available for storing items. An egg carton is a familiar example of a container with a maximum capacity.

The operations listed in Table 14-2 might lead you to wonder why programmers don't simply use an array whenever they need a collection in their applications. As you know from Chapter 12, you can perform similar operations on arrays. In fact, programmers had to use arrays exclusively in early programming languages because arrays were the only available collection type. But language designers have improved upon this situation. Consider, for example, a type of collection already discussed in this book, the string. A programmer who uses a string wants to take a logical perspective on it without regard to its underlying representation. Logically, a string is a sequence of characters with a specific set of operations or methods that apply to it. Some of these methods, such as `toLowerCase`, do not apply to arrays generally. Moreover, strings in Java are immutable, meaning that their component elements cannot be inserted, replaced, or removed, whereas that is not true of arrays. Finally, although a string might naturally be implemented as an array of characters, this is not the only possible implementation.

In modern languages like Java, arrays are considered a lower-level data structure and are often used to implement other collections. These collections, in turn, are more abstract and specialized, and, therefore, they are easier to use than arrays in many situations.

14.2 Java Collection Resources

Java's built-in collections are defined in the package `java.util`. This package includes two types of resources:

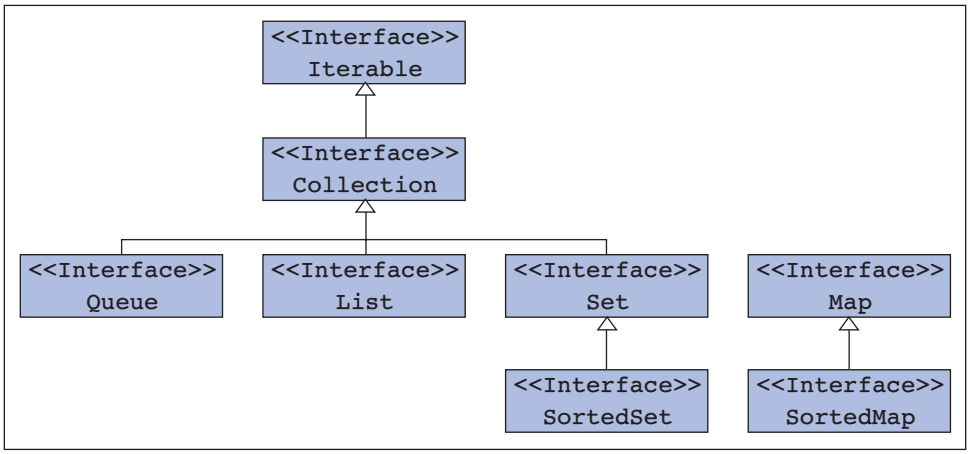
- A set of interfaces that specifies the operations for different types of collections
- A set of classes that implements these interfaces

As you learned in Chapter 11, an interface, such as `Pen` or `Shape`, provides a logical view of the behavior of a class of objects. A class, such as `RainbowPen` or `Circle`, realizes that behavior with an implementation. The same relationship between an interface and implementing class holds true in the case of collections. For example, `ArrayList` and `LinkedList` implement the `List` interface. Programmers who use lists develop their program logic with the methods specified in the `List` interface. They then use other criteria, such as run-time performance in a given set of circumstances, to select an implementing class. In the rest of this section, we summarize Java's collection interfaces and classes.

The Java Collection Interfaces

The `java.util` package includes interfaces for lists, sets, maps, sorted sets, sorted maps, and queues. These interfaces are organized in a hierarchy that includes two more general interfaces, called `Collection` and `Iterable`, as shown in Figure 14-1.

FIGURE 14-1
The `java.util` collection interfaces



Note that `Set` and `List` both extend `Collection`. That means that the set and list classes must implement methods, such as `size` and `isEmpty`, that are included in the `Collection` interface. Other methods, such as the index-based `remove`, are specific only to lists and so they are included only in the `List` interface. Oddly, the `Map` interface does not extend the `Collection` interface, but perhaps that is because maps have so little in common with lists and sets.

The `Collection` interface in turn extends the `Iterable` interface. This interface requires every implementing class to include an `iterator` method. The presence of this method allows the programmer to traverse any implementing collection with an enhanced `for` loop.

Once again, you must familiarize yourself with the methods in an interface to use any particular class that implements it. We explore the different collection interfaces in more detail later in this chapter.

The Java Collection Classes

The `java.util` package also includes implementing classes for lists, sets, maps, sorted sets, sorted maps, and queues. Figures 14-2 and 14-3 add some of these classes to the interfaces shown in Figure 14-1.

FIGURE 14-2
The `java.util` list and stack classes

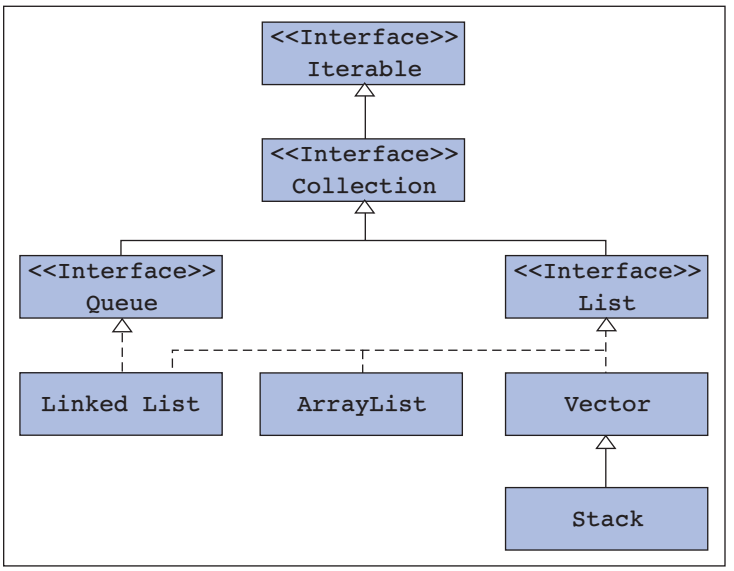
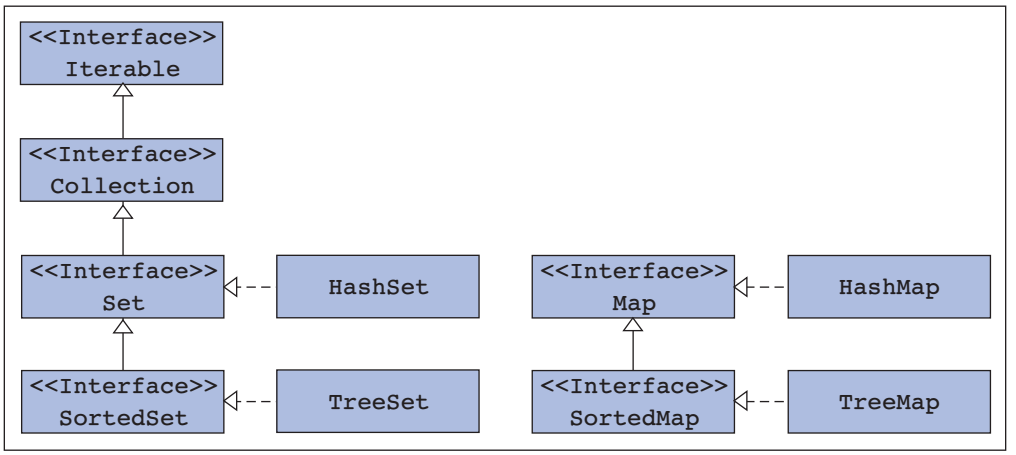


FIGURE 14-3
The `java.util` set and map classes



Note that there is just one implementing class for each of the set and map interfaces. However, the `List` interface has four implementing classes: `ArrayList`, `LinkedList`, `Vector`, and `Stack`. The `Stack` class implements the `List` interface by virtue of being a subclass of the `Vector` class. The `LinkedList` class also implements the `Queue` interface, which requires it to realize the behavior of a queue as well as a list.

The programmer chooses a collection class based on two criteria, the logic of an application and the collection's run-time performance. For example, if a program requires a list, the programmer would choose a class from among those that implement the `List` interface (the logic criterion). If this program frequently accesses elements at arbitrary index positions, the programmer would choose the `ArrayList` or `Vector` class because these two implementations of lists behave most like arrays (according to the run-time performance criterion).

In the following sections, we explore the run-time performance of various collection classes in more detail.

14.3 Using Lists

Like an array, a ***list*** is an object that contains a sequence of elements that are ordered by an integer index position. Elements in both structures can be accessed and replaced at a given position. However, there are several ways in which a list is unlike an array:

The programmer must use methods rather than the subscript operator `[]` to manipulate elements in a list.

A list tracks both its logical size (the number of elements that are currently in it) and its physical size (the number of cells available to store elements).

When the programmer first creates a list, the list's logical size is 0. When the programmer inserts elements into a list, the list automatically updates its logical size and adds cells to accommodate new objects if necessary. The list's logical size is also updated when an element is removed.

The index positions available for access in a list range from 0 to its logical size minus 1.

The List Interface

Lists recognize a very large number of methods. Table 14-3 includes some of those that are most commonly used. Some are specified in the `Collection` interface and others come from the `List` interface.

TABLE 14-3
Some methods in the `List` interface

METHOD	WHAT IT DOES
`boolean isEmpty()`	Returns `true` if the list contains no elements or `false` otherwise.
`int size()`	Returns the number of elements currently in the list.
`E get(int index)`	Precondition: 0 <= `index` < size of list. Returns the element at `index`.
`E set(int index, E element)`	Precondition: 0 <= `index` < size of list. Replaces the element at `index` with `element` and returns the old element.
`void add(int index, E element)`	Inserts the specified element at the specified position in this list (optional operation).
`E remove(int index)`	Precondition: 0 <= `index` < size of list. Removes and returns the element at `index`.
`boolean remove(Object element)`	Attempts to remove `element`. Returns `true` if `element` exists in the list and is removed or `false` otherwise.
`int indexOf(Object element)`	Returns the index of `element` or –1 if it's not found.
`boolean add(Object element)`	Adds `element` to the end of the list and returns `true` if successful or `false` otherwise.
`Iterator<E> iterator()`	Returns an iterator on the list, allowing the programmer to traverse it with an enhanced `for` loop.

The methods `get`, `set`, and the first `add` and `remove` methods are index-based. This means that these methods access a given index position in the list to perform their tasks. The second `remove` method and the `indexOf` method are element-based. Element-based methods search for a given element in the list to perform their tasks. The second `add` method appears to be element-based, but actually places the new element at the end of the list.

The methods in the `List` interface show an element type named `E` or `Object`, to indicate that the element type is generally an object of some kind or other. Recall that lists can contain elements of any type. However, as we shall see shortly, a particular element type must be specified when list variables are declared and when list objects are instantiated. When a name such as `E` is used to specify a type in an interface, it is called a ***type variable***. When a name such as `String` takes the place of the type variable during instantiation, it is called a ***type parameter***.

Declaring and Instantiating a List

A list is an object and must be instantiated, as in the following example, which creates an array list of strings:

```
import java.util.*;

List<String> list1 = new ArrayList<String>();
```

Note the following points:

- The `list1` variable is of type `List`, not `ArrayList`. Generally, it is a good idea to use the names of interfaces for type names wherever possible. This allows you to change the implementing class in your code without changing any of the code that uses your list variables.
- The programmer specifies no initial physical length, as is done with arrays. An array list has a default physical length, but the programmer has no need to know what it is.
- Note also the use of the type parameter `<String>` in both the variable declaration and the use of the constructor. The type parameter restricts the elements in this list to be of type `String`.

The general syntax for declaring a list variable and initializing it as a list object is as follows:

```
List<element-type> variable-name = new list-class-name<element-type>
  (parameters);
```

Like other classes, some collection classes include constructors that expect parameters. For example, the statement

```
List<String> list2 = new ArrayList<String>(list1);
```

creates a new list object containing the elements of `list1` and assigns it to the variable `list2`.

Using List Methods

The programmer manipulates a list by sending it messages. As mentioned earlier, there are methods for examining a list's logical size; testing it for emptiness (it's never full, at least in theory); inserting, removing, examining, or replacing elements at given positions; and searching for a given element, among others. For details on the other methods, consult Sun's documentation.

Let us extend the earlier code segment by loading the list with several strings and then displaying them in the terminal window:

```
import java.util.*;

List<String> list = new ArrayList<String>();

for (int i = 0; i < 5; i++)                  // list1 contains
   list.add(i, "Item" + (i + 1));           // Item1 Item2 Item3 Item4 Item5

for (int i = 0; i < list.size(); i++)                  // Display
   System.out.println(list.get(i));                    // Item1
                                                       // Item2
                                                       // Item3
                                                       // Item4
                                                       // Item5

// Or use an enhanced for loop
for (String str : list)                                // Display
   System.out.println(str);                            // Item1
                                                       // Item2
                                                       // Item3
                                                       // Item4
                                                       // Item5
```

The next code segment performs some sample searches:

```
System.out.println(list.indexOf("Item3"));     // Displays 2
System.out.println(list.indexOf("Martin"));    // Displays -1
```

Our final code segment removes the first element from the list and displays that element and the list's size after each removal, until the list becomes empty:

```
while (! list.isEmpty()){
   String str = list.remove(0);
   System.out.println(str);
   System.out.println("Size: " + list.size());
}
```

Lists and Primitive Types

The list is a powerful collection. There is one other restriction on its use, however. Unlike an array, a list can contain only objects, not values of primitive types, such as `int` and `double`. We explore a way of working around this restriction in the next subsection.

Primitive Types and Wrapper Classes

As mentioned earlier, Java distinguishes between primitive data types (numbers, characters, Booleans) and objects (instances of `String`, `Employee`, `Student`, and so on). Arrays can contain either primitive data values or objects, as in the following:

```
int x;                    // An integer variable
int nums[];               // An array of integers
```

```
Student student;           // A Student variable
Student students[];        // An array of Students
```

The elements in a list must all be objects. Thus, the following attempt to a create list of `int` fails with a syntax error:

```
List list<int> list1 = new ArrayList<int>(); // Invalid (syntax error)
```

A feature called ***wrapper classes*** allows us to store primitive data types in lists. A wrapper class is a class that contains a value of a primitive type. Values of the types `boolean`, `char`, `int`, and `double` can be stored in objects of the wrapper classes `Boolean`, `Character`, `Integer`, and `Double`, respectively. The following code segment shows how a wrapper class can be manipulated directly:

```
Integer intObj3 = new Integer(3);        // An Integer object
                                         // containing 3
Integer intObj4 = new Integer(4);        // An Integer object containing 4
int x = intObj3.intValue();              // Extracts 3 and saves in x
System.out.println(intObj3);             // Displays 3 using toString()
System.out.println(intObj3.equals(intObj4));      // Displays false
System.out.println(intObj3.compareTo(intObj4));   // Displays a negative
                                                  // number
```

Fortunately, you do not have to bother with these details to use primitive types with lists. Lists automatically "box" and "unbox" primitive values when they are used with list methods. The only requirement is that you use the appropriate wrapper class name as the element type parameter when declaring the list variable and instantiating the list. The next program example shows some manipulations of a list of integers:

```
// Example 14.1: Test an array list of integers

import java.util.*;

public class TestArrayList{

   public static void main(String[] args){
      // Create a list of Integers
      List<Integer> list = new ArrayList<Integer>();

      // Add the ints 1-100 to the list
      for (int i = 1; i <= 100; i++)
         list.add(i);

      // Increment each int in the list
      for (int i = 0; i < list.size(); i++)
         list.set(i, list.get(i) + 1);

      // Display the contents of the list
      for (int i : list)
         System.out.println(i);
   }
}
```

This code works because the methods `add` and `set` automatically box an `int` value into an `Integer` object, whereas the method `get` automatically unboxes an `Integer` object into an `int` value. Moreover, the enhanced `for` loop at the end of the program also unboxes each `Integer` object into an `int`.

Iterators

An ***iterator*** is an object that allows the programmer to visit all of the elements in a collection. The simplest type of iterator supports the methods in the `java.util.Iterator` interface, as shown in Table 14-4.

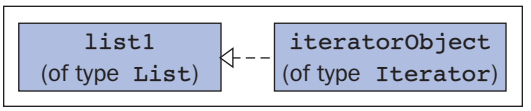
TABLE 14-4
The methods in the `Iterator` interface

METHOD	WHAT IT DOES
`boolean hasNext()`	Returns `true` if there are more elements to be visited, or `false` otherwise.
`E next()`	*Precondition*: `hasNext` returns `true`. Returns the next element and moves the position pointer ahead by one position.
`void remove()`	*Precondition*: `next` has just been run successfully. Removes from the collection the element most recently returned by `next`.

The programmer opens an iterator object on a collection by running the collection's `iterator` method. The next code segment accomplishes this with a list of strings named `list1`. The relationship between the iterator and the list is shown in Figure 14-4.

```
Iterator<String> iteratorObject = list1.iterator();
```

FIGURE 14-4
An iterator opened on a list object



At this point, the iterator's current position pointer is placed immediately before the first object in the collection, if the collection is not empty. The programmer can now test to determine if there are elements to visit by sending the iterator the `hasNext()` message. If this message returns `true`, it's safe to visit the first element in the collection by sending the iterator the `next()` message. This message returns the element to the caller and advances the iterator's current position pointer just beyond the first element. The programmer can repeat these two steps to visit all of the elements in the collection. The next code segment does this to print all the strings in `list1`:

```
while iteratorObject.hasNext(){
   String s = iteratorObject.next();
   System.out.println(s);
}
```

When the Java compiler sees an enhanced `for` loop used with a collection, it generates code that opens an iterator on the collection and automatically uses it to carry out the loop process. Thus, the code

```
for (String s : list1)
   System.out.println(s);
```

is actually translated to the earlier code segment during compilation.

Because a `for` loop is simpler to use, programmers don't normally resort to the use of an explicit iterator unless a collection's elements need to be removed during a traversal.

Confusing Arrays and Lists

As mentioned earlier, lists are accessed using methods, whereas arrays are accessed using subscripts. Beginners often confuse the two structures and try to use a subscript with a list or a method with an array. These two kinds of errors, both caught by the compiler, are shown in the next code segment:

```
// Create an array containing 3, 4, and 5
int array[] = {3, 4, 5};

// Create a list and add 2, 3, and 4
List<Integer> list = new ArrayList<Integer>();
list.add(2); list.add(3); list.add(4);

// Syntax errors: use list methods with the array
for (int i = 0; i < array.size(); i++)
   System.out.println(array.get(i));

// Syntax errors: using a subscript and length variable with the list
for (int i = 0; i < list.length; i++)
   System.out.println(list[i]);
```

Should I Use Arrays or Lists?

Now that you have seen two options for storing sequences of elements, the array and the list, it's time to consider the reasons you might prefer using one structure rather than the other.

Both structures allow the programmer to store and access elements by specifying an integer index position. However, the list is by far the more powerful and easier to use of the two structures. There are two reasons this is so:

- The list includes methods for a variety of tasks, such as insertions, removals, and searches. By contrast, an array provides just a subscript operation. As you saw in Chapter 12, the user of an array must implement higher-level operations in complex code.
- The list tracks its own logical size and grows or shrinks automatically as needed. By contrast, an array always has a fixed physical size that's independent of its logical size. This forces the user of an array to keep track of its logical size and to instantiate a larger array when needed, followed by copying elements from the old array to the new, and finally disposing of the old array.

These comparisons might lead you to conclude that you should never use an array in a program when a list is available. However, arrays are needed to implement higher-level data structures, such as grids (discussed in Chapter 12) and the array list itself.

There are at least two other situations in which an array might be on at least an equal footing with a list:

- You know the exact number of data elements to be inserted in advance, and they are all inserted at start-up and are never removed. In this case, the array is always full and is fairly easy to process.
- The operations on the sequence are more specialized than those provided by a list, or they consist of simple traversals. In these cases, the programmer really only must choose between using the methods `get` and `set` with a list and using the subscript with an array (or entirely ignoring them in an enhanced `for` loop).

In any case, because programmers have used arrays ever since the first high-level languages were developed, you will likely see arrays in code at one point or another in your programming experience.

Linked Lists

Thus far in this section, we have used `ArrayList` as the list class in our examples. The `LinkedList` class also implements the `List` interface. Therefore, it can be used wherever an `ArrayList` is used, as shown in the following code segment:

```
// Create an array list and use it
List<String> list = new ArrayList<String>();
// Use any of the List methods to process this list

// Now reset the list to a LinkedList
list = new LinkedList<String>();
// As before, use any of the List methods to process this list
```

Remember that array lists and linked lists are logically the same. However, they differ in their run-time performance characteristics. The index-based operations on array lists run in constant time, whereas the same operations on linked lists run in linear time. Thus, applications that require frequent `get` or `set` operations or index-based insertions or removals on lists should use `ArrayList` rather than `LinkedList`.

Because the `get` method runs in linear time on a linked list, an index-based loop to traverse a linked list runs in quadratic time! However, an enhanced `for` loop, which implicitly uses an iterator, runs in linear time on both array lists and linked lists.

All of the other `List` methods have similar run-time performance for `ArrayList` and `LinkedList`. However, the `LinkedList` class includes some additional methods not found in the `List` interface or the `ArrayList` class. These methods support accesses, insertions, and removals at either end of the list, and all operate in constant time.

EXERCISE 14.3

1. Write a method sum that expects a `List<Integer>` as a parameter. The method returns an `int` representing the sum of the integers in the list.

2. Write an index-based loop that prints the contents of a list.

EXERCISE 14.3 Continued

3. Compare the running times of the loop in Exercise 2 for an array list and a linked list, and write a code segment that guarantees the best running time for all list implementations.

4. Write a code segment that uses an iterator to remove all of the elements from a list named list.

Case Study 1: Building a Deck of Cards

In this case study, we show how to build resources that are used in an important class of applications—online card games! (*Note*: The authors do not endorse gambling.) We discuss the analysis, design, and implementation of a deck of cards that can be used in various types of card games. The development of programs for the games themselves is left as exercises.

Request

Develop resources for a deck of cards that can be used in online games.

Analysis

You have probably played some card games, such as Rummy, Crazy Eights, and Hearts. Most computer operating systems include one or more card games for entertaining diversion. Common examples are FreeCell and Solitaire on Windows systems. Most card games use a deck of cards with the following attributes:

- There are 52 cards in a deck.
- Each card has a suit and a rank.
- There are four suits: Spades, Hearts, Diamonds, and Clubs.
- Each suit has a color: black (Spades and Clubs) or red (Hearts and Diamonds).
- The cards can be ordered from highest to lowest by rank, as follows: King, Queen, Jack, 10, 9,..., 2. An Ace has either the highest or the lowest rank, depending on the game.
- The cards can also be ordered by suit, as follows: Spades, Hearts, Diamonds, Clubs.
- A deck has one card of each rank and each suit, which equals 52 (13 * 4) total cards.
- A card is face up if you can see its suit and rank; otherwise, it is face down.

Cards and a deck can be manipulated in the following ways:

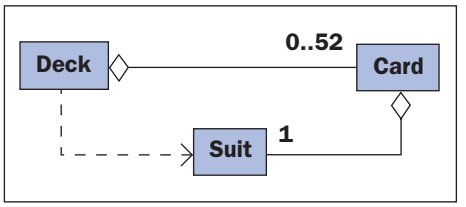
- Cards can be dealt or transferred from a deck to players.
- A deck can be shuffled, so that the cards can be dealt in random order.
- A card can be turned face up or face down.
- Two cards can be compared for equality, greater than, or less than.
- A card's suit, rank, and color can be examined.

The user interface for this case study consists of several tester programs for the various classes. Modern online card games also typically have flashy graphics that display images of the cards and allow the user to move them around by manipulating a mouse. We defer that aspect to an exercise and for now use a terminal-based interface to develop and illustrate the requirements for our deck of cards.

Classes

The two obvious classes suggested by our discussion of a deck of cards are `Deck` and `Card`. A deck contains 0 to 52 cards. It will be convenient to have one other class, called `Suit`, to represent the suit that each card has. The relationships among these classes are shown in the UML diagram of Figure 14-5.

FIGURE 14-5
The relationships among the classes `Deck`, `Card`, and `Suit`



The interfaces of the `Deck` and `Card` classes are listed in Tables 14-5 and 14-6. The `Card` methods require no comment. Because a deck decreases in size as cards are dealt, we include two `Deck` methods, `size()` and `isEmpty()`, to check on the deck's status. There are two methods for dealing cards. `deal()` returns a single card or `null` if the deck is empty. `deal(anInt)` is used to deal a hand of one or more cards all at once. If the size of the deck is greater than or equal to the integer parameter, this method returns an array of that number of cards. Otherwise, the method returns `null`. The method `shuffle` does nothing if the deck is not full. The method `reset` is used to gather the cards back into the deck to make it full.

The following is a short tester program that creates a deck, shuffles it, deals each card, and displays each card's information:

```
// Case Study 14.1: Test a deck

public class TestDeck{

   public static void main(String[] args){
      Deck deck = new Deck();
      deck.shuffle();
      int count = 0;
      while (! deck.isEmpty()){
         count++;
         Card card = deck.deal();
         System.out.println(count + ": " + card);
      }
   }
}
```

TABLE 14-5
The interface for the `Deck` class

DECK METHOD	WHAT IT DOES
`Deck()`	Constructor; creates a deck of 52 cards, unshuffled
`void reset()`	Restores the deck to its initial state
`void shuffle()`	If the deck is full, shuffles it
`boolean isEmpty()`	Returns `true` if the deck is empty or `false` otherwise
`int size()`	Returns the number of cards in the deck
`Card deal()`	If the deck is not empty, removes and returns a card; otherwise, returns `null`
`Card[] deal(int n)`	If the deck's size is greater than or equal to *n*, removes and returns *n* cards in an array; otherwise, returns `null`
`String toString()`	Returns a string representation of the deck

TABLE 14-6
The interface for the `Card` class

CARD METHOD	WHAT IT DOES
`Card(Suit suit, int rank)`	Constructor; creates a card with the specified suit and rank
`int getRank()`	Returns the card's rank
`Suit getSuit()`	Returns the card's suit
`boolean isFaceUp()`	Returns `true` if the card's face is up or `false` otherwise
`boolean isRed()`	Returns `true` if the card's suit color is red or `false` if it's black
`void turn()`	If the card is face up, makes it face down; or if it's face down, makes it face up
`boolean equals(Object other)`	Comparison for equality by rank
`int compareTo(Object other)`	Comparison for order of rank
`String toString()`	Returns a string representation of the card

From the client's perspective, `Suit` objects should be ordered consecutively like integers but named like strings. The strings "Clubs," "Diamonds," "Hearts," and "Spades" happen to have the appropriate natural ordering, so you might wonder why we would bother to define a new class for this purpose. The reason it is better to do so is suits should be restricted to four distinct values, whereas there are an infinite number of possible string values. Defining a `Suit` class also gives us an opportunity to introduce an important design technique that we discuss shortly. The interface of the `Suit` class includes four static constants and the methods `compareTo` and `toString`, as listed in Table 14-7.

TABLE 14-7
The interface for the `Suit` class

SUIT CONSTANT	WHAT IT DOES
`static final Suit spade`	The Suit value for all spades
`static final Suit heart`	The Suit value for all hearts
`static final Suit diamond`	The Suit value for all diamonds
`static final Suit club`	The Suit value for all clubs
`int compareTo(Object other)`	The usual behavior; comparison for order: spade > heart > diamond > club
`String toString()`	Returns a string representation of the suit

Note that there is no constructor in the interface. The only instances of `Suit` that ever exist are those named by the constants `Suit.spade`, `Suit.heart`, `Suit.diamond`, and `Suit.club`.

The next code segment is a short tester program that shows how suits are used.

```
// Case Study 14.1: Test the suits

public class TestSuit{

   public static void main(String[] args){

      Suit s = Suit.spade;
      Suit h = Suit.heart;
      Suit d = Suit.diamond;
      Suit c = Suit.club;

      // Display "spades hearts diamonds clubs"
      System.out.println(s + " " + h + " " + d + " " + c);

      System.out.println(s.equals(s));          // Display true
      System.out.println(s.equals(h));          // Display false
      System.out.println(s.compareTo(s));       // Display 0
      System.out.println(s.compareTo(d));       // Display 2
      System.out.println(d.compareTo(s));       // Display -2
   }
}
```

Our final tester program uses suits to build and manipulate some cards:

```
// Case Study 14.1: Test some cards

public class TestCard{

   public static void main(String[] args){
      Card queenSpades = new Card(Suit.spade, 12);
      Card jackClubs = new Card(Suit.club, 11);
```

```
        Card twoHearts = new Card(Suit.heart, 2);
        Card twoDiamonds = new Card(Suit.diamond, 2);

        // Display "Queen of spades"
        System.out.println(queenSpades);
            // Display true
            System.out.println(twoDiamonds.equals(twoHearts));
            // Display 1
            System.out.println(queenSpades.compareTo(jackClubs));
    }
}
```

Design and Implementation of `Card`

The design of the `Card` class calls for little comment. Note just three points in the implementation that follows. First, `Card` implements the `Comparable` interface, allowing cards to be used wherever comparisons are made. Second, in the method `isRed()`, suits are compared for equality using == rather than `equals`. We can get away with this because there is never more than one instance of each of the four suits. Third and finally, the method `turn()` flips the card by inverting or logically negating the `boolean` variable `faceUp`.

```
// Case Study 14.1: The Card class

public class Card implements Comparable{

   private Suit suit;
   private int rank;
   private boolean faceUp;

   public Card(Suit suit, int rank){
      this.suit = suit;
      this.rank = rank;
      faceUp = false;
   }

   public boolean equals(Object other){
      if (this == other)
         return true;
      else if (! (other instanceof Card))
         return false;
      else{
         Card otherCard = (Card)other;
         return rank == otherCard.rank;
      }
   }

   public int compareTo(Object other){
      if (! (other instanceof Card))
         throw new IllegalArgumentException("Parameter must be a Card");
      Card otherCard = (Card)other;
      return rank - otherCard.rank;
   }
```

```
    public int getRank(){
       return rank;
    }

    public Suit getSuit(){
       return suit;
    }

    public boolean isFaceUp(){
       return faceUp;
    }

    public boolean isRed(){
       return suit == Suit.heart || suit == Suit.diamond;
    }

    public void turn(){
       faceUp = ! faceUp;
    }

    public String toString(){
       return rankToString() + " of " + suit;
    }

    private String rankToString(){
       if (rank == 1)
          return "Ace";
       else if (rank == 11)
          return "Jack";
       else if (rank == 12)
          return "Queen";
       else if (rank == 13)
          return "King";
       else
          return "" + rank;
    }
}
```

Design and Implementation of `Deck`

According to the requirements established during analysis, a deck contains from 0 to 52 cards. Thus, an array list of type `Card` is an appropriate choice of a data structure to hold the cards. All of the `Deck` methods manage the list of cards for the client. The only method that is complex enough to warrant explicit design is `shuffle()`. A standard way to shuffle real cards is to split the deck and merge the two sets of cards in such a manner that they roughly interleave (the first card of the second half comes after the first card of the first half, and so forth). The development of this algorithm is left as an exercise. Instead, we adopt a simpler method: Create a temporary array of 52 positions; remove each card from the list and place it at a random, unoccupied position of the array; and when the list becomes empty, transfer the cards from the array back to the list. Here is the pseudocode for this process:

```
Set array to a new array of 52 cells
While the list is not empty do
```

```
      Set card to the last card in the list
      Set index to a random number between 0 and 51
      While array[index] != null
         Set index to a random number between 0 and 51
      Set array[index] to card
   For each card in array
      Add the card to the list
```

Note that the nested loop uses random integers to locate an array position that contains `null`. All of the array's cells are `null` when the array is instantiated. A `null` array cell indicates an empty slot in which to insert the next card. Here is a complete listing of the `Deck` implementation:

```
// Case Study 14.1: The Deck class

import java.util.*;

public class Deck{

   public static final int MAX_SIZE = 52;

   private List<Card> cards;

   public Deck(){
      reset();
   }

   public void reset(){
      // Create a new list and add 13 cards from each suit
      cards = new ArrayList<Card>();
      addSuit(Suit.spade);
      addSuit(Suit.heart);
      addSuit(Suit.diamond);
      addSuit(Suit.club);
   }

   // Helper method to add 13 cards from a single suit
   private void addSuit(Suit suit){
      for (int i = 1; i <= 13; i++)
         cards.add(new Card(suit, i));
   }

   public boolean isEmpty(){
     return cards.isEmpty();
   }

   public int size(){
      return cards.size();
   }

   public Card deal(){
      if (isEmpty())
         return null;
```

```
        else
            return cards.remove(cards.size() - 1);
    }

    public Card[] deal(int number){
        if (number > cards.size())
            return null;
        else{
            Card[] hand = new Card[number];
            for (int i = 0; i < hand.length; i++)
                hand[i] = deal();
            return hand;
        }
    }

    public void shuffle(){
        if (cards.size() < MAX_SIZE)
            return;
        Random gen = new Random();
        // Remove cards from the list and place at random positions
        // in an array
        Card[] array = new Card[MAX_SIZE];
        while (cards.size() > 0){
          Card card = cards.remove(cards.size() - 1);
          int i = gen.nextInt(MAX_SIZE);
          while (array[i] != null)
              i = gen.nextInt(MAX_SIZE);
          array[i] = card;
        }
        // Transfer the shuffled cards back to the list
        for (Card card : array)
            cards.add(card);
    }

    public String toString(){
        String result = "";
        for (Card card : cards)
            result += card + "\n";
        return result;
    }
}
```

Design and Implementation of `Suit`

The purpose of the `Suit` class is to provide exactly four distinct instances that represent the suits Spades, Hearts, Diamonds, and Clubs. These objects can be compared for equality and for their relative ordering. As we mentioned earlier, the class's interface includes four static constants for these objects, as well as the methods `compareTo` and `toString`. Equality is determined with the operator ==.

The design of the class is unusual, in that clients cannot instantiate it. However, four instances are indeed created when the static constants are initialized (at program start-up). Each instance has an integer that determines its rank in the ordering of suits and a name that

is used for its string representation. A private constructor initializes these variables when the instances are created and assigned to the constants. A listing of the `Suit` class follows.

```
// Case Study 14.1: A Suit class

public class Suit implements Comparable{

   static public final Suit spade   = new Suit(4, "spades");
   static public final Suit heart   = new Suit(3, "hearts");
   static public final Suit diamond = new Suit(2, "diamonds");
   static public final Suit club    = new Suit(1, "clubs");

   private int order;
   private String name;

   private Suit(int ord, String nm){
      name = nm;
      order = ord;
   }

   public int compareTo(Object other){
      if (! (other instanceof Suit))
         throw new IllegalArgumentException("Parameter must be a Suit");
      Suit otherSuit = (Suit)other;
      return order - otherSuit.order;
   }

   public String toString(){
      return name;
   }
}
```

14.4 Using Stacks

A *stack* is a sequence of elements in which access is completely restricted to just one end, called the top. The classic example is the stack of clean trays found in every cafeteria. Whenever a tray is needed, it is removed from the top of the stack, and whenever clean ones come back from the dishwasher, they are again placed on the top. No one ever takes some particularly fine tray from the middle of the stack, and it is even possible that trays near the bottom are never used. Stacks are said to adhere to a Last In First Out protocol (LIFO). The last tray taken out of the dishwasher is the first one taken by a customer. Although we continually add more papers to the top of the piles on our desks, these piles do not quite qualify because we often need to remove a long-lost paper from the middle. With a genuine stack, the item we get next is always the one added most recently.

The `Stack` Class

The `java.util` package includes a `Stack` class that is a subclass of `Vector`. As we saw earlier, the `Vector` class implements the `List` and `Collection` interfaces. This means that programmers can treat stacks as if they were vectors by inserting, replacing, or removing an element at any position, thus violating the LIFO spirit of a stack. For now, we ignore this nicety and restrict

our attention to the set of methods that ought to be used with the `Stack` class. These methods, some of which are included in the `Vector` class and others of which are included in the `Stack` class, are listed in Table 14-8.

TABLE 14-8
Stack methods

METHOD	WHAT IT DOES
`boolean isEmpty()`	Returns `true` if the stack contains no elements or `false` otherwise.
`int size()`	Returns the number of elements currently in the stack.
`E peek()`	*Precondition*: The stack is not empty. Returns the element at the top of the stack.
`E pop()`	*Precondition*: The stack is not empty. Removes and returns the element at the top of the stack.
`void push(E element)`	Adds `element` to the top of the stack.

As with other collections, a stack's element type should be specified when it is created. A simple example illustrates the use of the `Stack` class. In this example, the strings in a list are transferred to a stack and then displayed. Their display order is the reverse of their order in the list. The last line of code attempts to pop from an empty stack, which throws an `EmptyStackException`.

```
// Example 14.2: Tests a stack

import java.util.*;

public class TestStack{

  public static void main(String[] args){
      // Create a list and add some strings
      List<String> lst = new ArrayList<String>();
      for (int i = 0; i <= 5; i++)
         lst.add("String " + i);

      // Create a stack and transfer the strings
      Stack<String> stk = new Stack<String>();
      for (String str : lst)
         stk.push(str);

      // Pop and display objects from the stack
      while (! stk.empty())
         System.out.println(stk.pop());

      // Cause an EmptyStackException
      stk.pop();
   }
}
```

Applications of Stacks

Applications of stacks in computer science are numerous. Following are just a few examples:

- Analyzing and checking the syntax of expressions in programming languages—a problem in compiler design
- Translating infix expressions to postfix form
- Evaluating postfix expressions—discussed later in this chapter
- Backtracking algorithms used in problems such as automated theorem proving and game playing
- Managing computer memory in support of method calls
- Supporting the "undo" feature in text editors, word processors, spreadsheet programs, drawing programs, and similar applications
- Maintaining a history of the links visited by a Web browser

Example: Evaluating Postfix Expressions

In daily life, we are so accustomed to evaluating simple arithmetic expressions that we give little thought to the rules involved. So you might be surprised by the difficulty of writing an algorithm to do the same thing. An indirect approach to the problem works best. First, you transform an expression from its familiar infix form to a postfix form, and then you evaluate the postfix form. In the infix form, each operator is located between its operands, whereas in the postfix form, an operator immediately follows its operands. Table 14-9 gives several simple examples.

TABLE 14-9
Some infix and postfix expressions

INFIX FORM	POSTFIX FORM	VALUE
34	34	34
34 + 22	34 22 +	56
34 + 22 * 2	34 22 2 * +	78
34 * 22 + 2	34 22 * 2 +	750
(34 + 22) * 2	34 22 + 2 *	112

There are similarities and differences between the two forms. In both, operands appear in the same order. However, the operators sometimes do and sometimes do not. The infix form sometimes requires parentheses; the postfix form never does. Infix evaluation involves rules of precedence; postfix evaluation applies operators as soon as they are encountered. For instance, consider the steps in evaluating the infix expression 34 + 22 * 2 and the equivalent postfix expression 34 22 2 * +.

Infix evaluation: 34 + 22 * 2 → 34 + 44 → 78

Postfix evaluation: 34 22 2 * + → 34 44 + → 78

The use of parentheses and operator precedence in infix expressions is for the convenience of the human beings who read them and write them. By eliminating the parentheses and ignoring operator precedence, the equivalent postfix expressions present a computer with a format that is much easier and more efficient to evaluate.

Evaluation of postfix expressions consists of three steps:

1. Scan across the expression from left to right.
2. On encountering an operator, apply it to the two preceding operands and replace all three by the result.
3. Continue scanning until reaching the expression's end, at which point only the expression's value remains.

To express this procedure as a computer algorithm, we use a stack of operands. In the algorithm, the term "token" refers to either an operand or an operator:

```
create a new stack
while there are more tokens in the expression
   get the next token
   if the token is an operand
      push the operand onto the stack
   else if the token is an operator
      pop the top two operands from the stack
      use the operator and the two operands just popped to compute a value
      push the resulting value onto the stack
   end if
end while
return the value at the top of the stack
```

Table 14-10 shows a trace of the algorithm as it is applied to the expression 4 5 6 * + 3 –.

TABLE 14-10
Tracing the evaluation of a postfix expression

PORTION OF POSTFIX EXPRESSION SCANNED SO FAR	OPERAND STACK	COMMENT
4	4	Push the operand 4.
4 5	4 5	Push the operand 5.
4 5 6	4 5 6	Push the operand 6.
4 5 6 *	4 30	Replace the top two operands by their product.
4 5 6 * +	34	Replace the top two operands by their sum.
4 5 6 * + 3	34 3	Push the operand 3.
4 5 6 * + 3 –	31	Replace the top two operands by their difference. Pop the final value.

Scanning Expressions

The first step in the process of evaluating postfix expressions, as discussed previously, is stated simply as "scan across the expression from left to right." Although this step sounds simple, its implementation is not. Its input datum is typically a line of text, represented as a string. In this string, the scanning algorithm must detect and extract two types of tokens, operators (such as "+" and "*") and operands (such as "88" and "56"). The latter must also be converted to the corresponding Java `int` values before further processing.

Fortunately, Java's `Scanner` class comes to the rescue, by providing the capability of recognizing and extracting individual words or tokens in a string of characters. The scanner is first created in the usual manner, except that the source string is passed as a parameter to the scanner's constructor. In the next code segment, a scanner is opened on the string "33 + 44 * 55."

```
Scanner tokens = new Scanner("33 + 44 * 55");
```

The scanner object assumes that the delimiters between tokens are blank spaces by default.

To extract the tokens from the string, you use the methods `hasNext` and `next` in a standard traversal loop, as follows:

```
while (tokens.hasNext()){
   String token = tokens.next();
   // Do something with token
}
```

To determine whether a given token represents an operand (an integer), you can examine its first character with the `Character.isDigit` method. The `Integer.parseInt` method can then be used to convert the token to the integer value that it represents. Here are the signatures of these two methods:

```
public static boolean isDigit(char ch)

public static int parseInt(String s)
```

EXERCISE 14.4

1. Assume that the stack `s` contains the sequence of values 30 25 44, and that 44 is at the top. Write the values contained in `s` after each of the following methods are run, in sequence:

a. `s.push(55);`

b. `s.pop();`

c. `s.pop();`

2. A palindrome is a string that reads the same in either direction (left to right or right to left). Write a code segment that uses a stack to determine whether or not a string is a palindrome. (*Hint*: The stack should contain characters.)

EXERCISE 14.4 Continued

3. Write the values of the following postfix expressions:
   a. 44 2 + 3 *
   b. 44 2 3 + *

4. Assume that a string named `source` contains arithmetic operators and integer operands. Write a code segment that scans across this string. This code should print each token in the string followed by that token's type ("operator" or "integer").

14.5 Using Queues

Like stacks, ***queues*** are sequences of elements. However, with queues, insertions are restricted to one end, called the rear, and removals are restricted to the other end, called the front. A queue thus supports a First In First Out (FIFO) access protocol. Queues are omnipresent in everyday life and occur in any situation where people or things are lined up for processing on a first-come, first-served basis. Checkout lines in stores, highway tollbooth lines, and airport baggage check-in lines are familiar examples of queues.

The `Queue` Interface and Its `LinkedList` Implementation

Queues have two fundamental operations: `add` (sometimes called `enqueue`), which adds an item to the rear of a queue, and `remove` (sometimes called `dequeue`), which removes and returns an item from the front. The `java.util` package includes a `Queue` interface. Like the `List` interface, the `Queue` interface extends the `Collection` interface, so there are many methods that are available to the user of a queue. The methods most commonly used with queues are shown in Table 14-11.

TABLE 14-11
Queue methods

METHOD	WHAT IT DOES
`boolean isEmpty()`	Returns `true` if the queue contains no elements or `false` otherwise.
`int size()`	Returns the number of elements currently in the queue.
`E peek()`	*Precondition*: The queue is not empty. Returns the element at the front of the queue.
`E remove()`	*Precondition*: The queue is not empty. Removes and returns the element at the front of the queue.
`boolean add(E element)`	Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning `true` upon success and throwing an `IllegalStateException` if no space is currently available.

The Java class that implements the `Queue` interface, the `LinkedList` class, also includes all of the methods in the `List` interface. Thus, it would seem that Java programmers might encounter the same issue using queues as they do with stacks: Some of the methods available, such as insertions or removals from the middle of a queue, violate the FIFO spirit of this collection. However, because the `LinkedList` class implements the `Queue` interface, the Java compiler can restrict the programmer's use of methods to those in `Queue` and `Collection`. All you have to do is use `Queue` as the type name of the queue variables, as shown in the following example:

```
// Example 14.3: Tests a queue

import java.util.*;

public class TestQueue{

public static void main(String[] args){
  // Create a queue of integers
  Queue<Integer> queue = new LinkedList<Integer>();
  // Add 5 integers to the queue
  for (int i = 1; i <= 5; i++)
     queue.add(i);

  // Traverse the queue to print the elements
  for (int i : queue)
     System.out.println(i);

  // Remove and print the element at the front
  System.out.println(queue.remove());

  // Try to replace the element at position 2 --- syntax error!
  System.out.println(queue.set(2, 10));
}
}
```

Because we use `Queue` as the type name of the variable, we can use the methods `add` and `remove` and also the enhanced `for` loop with it. However, when we try to use the `List` method `set` with this variable, the compiler triggers a syntax error, even though we know that the object referenced by this variable is a `LinkedList`. Thus, there is yet another reason to use interface names when declaring variables: They protect the programmer from doing things that should not be done with certain types of objects.

Applications of Queues

Most queues in computer science involve scheduling access to shared resources. Here are some examples:

- **CPU access**—Processes are queued for access to a shared CPU.
- **Disk access**—Processes are queued for access to a shared secondary storage device.
- **Printer access**—Print jobs are queued for access to a shared laser printer.

Queues are also used in computer simulations. Computer simulations are used to study the behavior of real-world systems, especially when it is impractical or dangerous to experiment with these systems directly. For example, a computer simulation could mimic traffic flow on a busy highway. Urban planners could then experiment with factors that affect traffic flow, such as the number and types of vehicles on the highway, the speed limits for different types of vehicles, the number of lanes in the highway, and the frequency of tollbooths. Outputs from such a simulation might include the total number of vehicles able to move between designated points in a designated period and the average duration of a trip. By running the simulation with many combinations of inputs, the planners could determine how best to upgrade sections of the highway, subject to the ever-present constraints of time, space, and money.

As a second example, consider the problem faced by the manager of a supermarket who is trying to determine the number of checkout cashiers to schedule at various times of the day. Some important factors in this situation are the following:

- The frequency with which new customers arrive
- The number of checkout cashiers available
- The number of items in a customer's shopping cart
- The period of time considered

These factors could be inputs to a simulation program, which would then determine the total number of customers processed, the average time each customer waits for service, and the number of customers left standing in line at the end of the simulated time period. By varying the inputs, particularly the frequency of customer arrivals and the number of available checkout cashiers, a simulation program could help the manager make effective staffing decisions for busy and slow times of the day. By adding an input that quantifies the efficiency of different checkout equipment, the manager can even decide whether it is more cost-effective to add more cashiers or buy better, more efficient equipment.

EXERCISE 14.5

1. Assume that the queue `q` contains the sequence of values 30 25 44, and that 44 is at the rear. Write the values contained in `q` after each of the following methods are run:
   a. `q.add(55)`
   b. `q.remove()`
   c. `q.add(13)`

2. Write a code segment that removes all of the elements from the stack `s` and adds them to the queue `q`.

3. Assume that the stack `s` contains the sequence of values 30 25 44, and that 44 is at the top. Write the values contained in the queue `q` after they are removed from the stack and added to the queue (the queue is empty before this process is run). The value at the left of the queue is at its front.

14.6 Using Sets

The concept of a ***set*** should be familiar from mathematics. A set is a collection of items that, from the client's perspective, are unique. That is, there are no duplicate items in a set. There are many operations on sets in mathematics. Some of the most typical are

- Test for the empty set.
- Return the number of items in the set.
- Add an item to the set.
- Remove an item from the set.
- Test for set membership (whether or not a given item is in the set).
- Return the union of two sets. The union of two sets A and B is a set that contains all of the items in A and all of the items in B.
- Return the intersection of two sets. The intersection of the two sets A and B is the set of items in A that are also items in B.
- Return the difference of two sets. The difference of two sets A and B is the set of items in A that are not also items in B.
- Test a set to determine whether or not another set is its subset. The set B is a subset of set A if and only if B is an empty set or all of the items in B are also in A.

To describe the contents of a set, we use the notation {<item1> . . . <item-n>} and assume that the items are in no particular order. Table 14-12 shows the results of some operations on sample sets.

TABLE 14-12
Results of typical set operations

SETS A AND B	UNION	INTERSECTION	DIFFERENCE	SUBSET
{12 5 17 6} {42 17 6}	{12 5 42 17 6}	{17 6}	{12 5}	False
{21 76 10 3 9} {}	{21 76 10 3 9}	{}	{21 76 10 3 9}	True
{87} {22 87 23}	{22 87 23}	{87}	{}	False
{22 87 23} {87}	{22 87 23}	{87}	{22 23}	True

The java.util.Set Interface

The most commonly used methods in the `Set` interface are listed in Table 14-13.

TABLE 14-13
Some `Set` methods

METHOD	WHAT IT DOES
`boolean isEmpty()`	Returns `true` if the set contains no elements or `false` otherwise.
`int size()`	Returns the number of elements currently in the set.
`boolean contains(Object element)`	Returns `true` if the set contains `element` or `false` otherwise.
`boolean remove(Object element)`	Removes `element` if it exists and returns `true`, or returns `false` if `element` does not exist.
`boolean add(Object element)`	Adds `element` to the set if `element` does not already exist and returns `true`. Returns `false` otherwise.
`Iterator<E> iterator()`	Returns an iterator on the set, allowing the programmer to traverse it with an enhanced `for` loop.

Note that sets support an `iterator` method. You might have been wondering how you can examine all the items in a set after they have been added, in view of the fact that there is no index-based access as with lists. You use an iterator, which supports the visiting of items in an unspecified order.

Applications of Sets

Aside from their role in mathematics, sets have many applications in the area of data processing. For example, in the field of database management, the answer to a query that contains the conjunction of two keys could be constructed from the intersection of the sets of items containing these keys.

Implementing Classes

The implementing classes for sets in `java.util` are called `HashSet` and `TreeSet`. The name `HashSet` derives from the technique, called ***hashing***, by which the set's items are accessed. Hashing supports close to constant time accesses, insertions, and removals of elements from a collection. The following code segment is a trivial illustration of the use of a `HashSet` of strings:

```
Set<String> s = new HashSet<String>();

s.add("Bill");
s.add("Mary");
s.add("Jose");
s.add("Bill");                          // Duplicate element
System.out.println(s.size());           // Prints 3
System.out.println(s.contains("Jose")); // Prints true
```

Now suppose we want to print the names in this set. By creating an iterator on the set, we can retrieve the names one by one and print them:

```
Iterator<String> iter = s.iterator();

while (iter.hasNext()){
   String name = iter.next();
   System.out.println(name);
}
```

An enhanced `for` loop works just as well and is simpler and foolproof:

```
for (String name : s)
   System.out.println(name);
```

As another example, let us define a `static` method that creates the intersection of two sets of strings. The intersection of two sets is a new set containing only the items the two sets have in common. Following is the code:

```
public static Set<String> intersection(Set<String> a, Set<String> b){
   Set<String> result = new HashSet<String>();
   for (String element : a)
      if (b.contains(element))
         result.add(element);
   return result;
}
```

The `TreeSet` class implements the `SortedSet` interface. Although this type of set is still an unordered collection, we can use an iterator or a `for` loop to visit its elements in sorted order. When elements are inserted, they must implement the `Comparable` interface, as described in Chapter 12. Unlike a `HashSet`, a `TreeSet` supports access to elements in logarithmic time.

EXERCISE 14.6

1. Write a code segment that removes all of the elements from a list `list` and adds them to a set `s`.
2. Assume that a list `list` contains the values 33 44 67 100 44. Which values does the set `s` contain after they are removed from `list` and added to `s`?
3. Write a code segment that prints the sum of the integers contained in the set `s`.
4. Define a `static` method `union` that expects two sets of integers as parameters. This method builds and returns the union of the two sets.

14.7 Using Maps

A *map*, sometimes also referred to as a *table*, is a collection in which each item, or value, is associated with a unique key. Users add, remove, and retrieve items from a map by specifying their keys. A map is also occasionally referred to as a *keyed list*, *dictionary*, or an *association list*. The package `java.util` defines several classes, `Dictionary`, `Hashtable`, `HashMap`, and `TreeMap`, which all represent maps. Another way to think of a map is as a collection of unique items called entries or associations. Each entry contains a key and a value (the item).

Table 14-14 shows the data in two maps. The first map is keyed by strings, and the second map is keyed by integers.

TABLE 14-14
A map keyed by strings and a map keyed by integers

MAP 1		MAP 2	
KEY	VALUE	KEY	VALUE
`"occupation"`	`"teacher"`	80	"Mary"
`"hair color"`	`"brown"`	39	"Joe"
`"height"`	72	21	"Sam"
`"age"`	72	95	"Lily"
`"name"`	`"Bill"`	40	"Renee"

Note the following points about the two sample maps:

- The keys are in no particular order.
- The keys are unique. That is, the keys for a given map form a set.
- The values need not be unique. That is, the same value can be associated with more than one key.

There are many operations that you could perform on maps. At a bare minimum, a client should be able to do the following:

- Test a map for emptiness.
- Determine a map's size.
- Insert a value at a given key.
- Remove a given key (also removing the associated value).
- Retrieve a value at a given key.
- Determine whether or not a map contains a key.
- Determine whether or not a map contains a value.
- Examine all of the keys.
- Examine all of the values.

Maps have a wide range of applications. For example, interpreters and compilers of programming languages make use of symbol tables. Each key in a symbol table corresponds to an identifier in a program. The value associated with a key contains the attributes of the identifier—a name, a data type, and other information. Perhaps the most prevalent application of maps is in database management.

The `java.util.Map` Interface

Table 14-15 lists and describes the most frequently used `Map` methods. In this table, the object associated with a key is called its value. A map uses two type variables, named `K` and `V`, to refer to the map's key type and value type, respectively.

TABLE 14-15
Some `Map` methods

METHOD	WHAT IT DOES
`boolean isEmpty()`	Returns `true` if the map contains no elements or `false` otherwise.
`int size()`	Returns the number of elements currently in the map.
`boolean containsKey(Object key)`	Returns `true` if the map contains `key` or `false` otherwise.
`V get(Object key)`	Returns the value associated with `key` if `key` exists or `null` otherwise.
`V remove(Object key)`	Removes and returns the value associated with `key` if `key` exists or `null` otherwise.
`V put(K key, V value)`	Adds `key/value` to the map if `key` does not already exist and returns `null`. Otherwise, replaces the value at `key` with `value` and returns the previous value.
`Set<K> keySet()`	Returns a set of the map's keys.
`Collection<V> values()`	Returns a collection of the map's values.

Note that the methods `keySet` and `values` allow the client to examine all of a map's keys and values, respectively. We discuss the `Collection` type and the idea of a collection view later in this chapter. For now, suffice it to say that you can open an iterator on a collection. The following code segment, which associates strings with integers in a map, shows how easy it is to visit all the keys and values in the map with iterators:

```
Map<String, Integer> map = new HashMap<String, Integer>();

// . . . add a bunch of keys and values to the map

// Obtain the views of the map's keys and values.
Set<String> s = map.keySet();
Collection<Integer> c = map.values();

// Open iterators on these views and display their contents
Iterator<String> keys = s.iterator();
```

```
while (keys.hasNext())
   System.out.println(keys.next());
Iterator<Integer> values = c.iterator();
while (values.hasNext())
   System.out.println(values.next());
```

Note that a map needs two type parameters when it is declared and instantiated. The first type is the key type and the second type is the value type. Occasionally, you will see nested type parameters when used with maps or other collections. For example, suppose we need a map of sets of strings, keyed by strings. The next code segment declares and instantiates such a map:

```
Map<String, Set<String>> map = new HashMap<String, Set<String>>();
```

Implementing Classes

The package `java.util` provides two implementations of the `Map` interface called `HashMap` and `TreeMap`. In the following sample code segment, we set up a `HashMap` whose keys are student IDs and whose values are the associated `Student` objects:

```
// Declare and instantiate a map
Map<String, Student> studentMap = new HashMap<String, Student>();

for (int i = 1; i <= 4; i++){
   Student stu = new Student();    // Instantiate a new student
   stu.setId("s" + i);             // Set the student's attributes
   stu.setName . . .               // . . .
   stu.setScore . . .              // . . .
   studentMap.put("s" + i, stu);   // Add the student to the map using
                                   // the student's ID as the key
}
```

Notice the following points about this code:

- We have added an attribute called ID to the `Student` class. Unlike a name, which might be duplicated, an ID is unique.
- A student ID is both one of its attributes and its key. It is common practice to choose an object's key from among its attributes.
- An object is retrieved from a map using the `get` method, as illustrated in this statement that retrieves and displays the student whose key is `"s1"`:

```
System.out.println(studentMap.get("s1"));
```

EXERCISE 14.7

1. Assume that the list `keys` contains the set of keys to be inserted into a map, and the list `values` contains the corresponding values. The keys are strings and the values are integers. Write a code segment that declares a map variable `m` of strings to integers, assigns this variable a new map object, and copies the data from the two lists to the map.

EXERCISE 14.7 Continued

2. Assume that a map `m` associates students' names with numeric grades. Write a code segment that prints the name of the student with the highest grade.

3. Several students could be tied for the highest grade in the map of Exercise 2. Write a code segment that uses a set to accumulate their names and print them.

Case Study 2: Interactive Psychotherapy

To illustrate the use of maps, we offer an amusing program that will allow you to earn a fortune as a nondirective psychotherapist.

Request

Write a program that emulates a nondirective psychotherapist. The practitioner of this kind of therapy is essentially a good listener who responds to a patient's statements by rephrasing them or indirectly asking for more information.

Analysis

Figure 14-6 shows the system's interface as it changes throughout a sequence of exchanges with the user.

FIGURE 14-6

A session with the therapist program

```
Good day. What is your problem?

Enter your response here or Q to quit: my teacher hates me

Why do you say that your teacher hates you

Enter your response here or Q to quit: she always calls on
the girls in the class

Please tell me more

Enter your response here or Q to quit: i would like to get
called on too

You seem to think that you would like to get called on too

Enter your response here or Q to quit: q
```

When the user enters a statement, the program responds in one of two ways:

1. With a randomly chosen hedge, such as "Please tell me more."

2. By changing some keywords in the user's input string and appending this string to a randomly chosen qualifier; thus, to "My teacher always plays favorites," the program might reply with "Why do you say that your teacher always plays favorites?"

The program consists of three classes: `TherapistApp`, `TherapistView`, and `Therapist`. A `TherapistView` object runs a sentinel-driven loop, passes user inputs to a `Therapist` object, and displays that object's reply. A `Therapist` object does the actual work of constructing a reply from the user's statement.

Design of the `TherapistView` Class

The `TherapistView` class creates a new `Therapist` object at program start-up, displays a prompt, and waits for user input. The following is pseudocode for the main loop:

```
output a greeting to the patient
while true
   prompt for and input a string from the patient
   if the string equals "Q" then break
   ask the therapist object for its reply to this string
   display the therapist's reply to the patient
```

Design of the `Therapist` Class

The `Therapist` class stores hedges and qualifiers in two sets and stores keywords and their replacements in a map. These collections are initialized when a therapist object is instantiated.

The behavior of a `Therapist` object is carried out by a set of cooperating methods. The top-level method is called `reply`. This method chooses one of two strategies for constructing a reply after generating a random number from 1 through 3. If that number is 1 (representing a ⅓ probability), `reply` generates and returns a hedge. Otherwise, the number is either 2 or 3 (representing ⅔ probability), and `reply` then changes the persons in the patient's input string, prepends a randomly chosen qualifier to the result, and returns it.

The tasks of constructing a hedge and changing persons in a sentence are delegated to the methods `hedge` and `changePerson`, respectively. The helper method `selectRandom` returns a string selected at random from a set. The helper method `findReplacement` returns either a replacement for a word if it exists or the word itself. The helper method `qualifier` returns a qualifier selected at random.

Following is pseudocode for these methods, which reflect a top-down design:

```
String reply(String patientString){
   pick a random number between 1 and 3
   if the number is 1
      call the hedge method to return a randomly chosen hedge
   else{
      call the qualifier method to randomly choose a qualifying phrase
   call the changePerson method to change persons in the patient's string
      return the concatenation of these two strings
   }
}

String hedge(Set set){
   randomly select a number between 0 and the size of the hedges set - 1
   use that number to return the string from the hedges set
```

```
}
String qualifier(Set set){
   randomly select a number between 0 and the size of the qualifiers set - 1
   use that number to return the string from the qualifiers set
}

String changePerson(String patientString){
   open a scanner on the patientString
   create an empty result string
   while the scanner has more tokens
      get the next token
      find the replacement word for that token
      append the replacement word to the result string
   }
   return the result string
}

String findReplacement(String word){
   if the word is a key in the replacements map
      return the value stored at that key
   else
      return the word
}

String selectRandom(Set set){
   generate a random number between 0 and one less than set's size
   use this number to locate a string in the set
   return the string
}

int randomInt(int low, int high){
   use Math.random to generate a random double 0 <= x < 1
   map x to an integer low <= i <= high
   return i
}
```

We are now ready to present the Java code for the classes.

Implementation of the `TherapistView` Class

```
// Case Study 14.2: View for the therapist program

import java.util.Scanner;

public class TherapistView{

   private Therapist therapist = new Therapist();

   public TherapistView(){
      Scanner reader = new Scanner(System.in);
      System.out.println("Good day. What is your problem?");
      while (true){
         System.out.print("\nEnter your response here or Q to quit: ");
         String patientString = reader.nextLine();
```

```
            if (patientString.equalsIgnoreCase("Q"))
            break;
            String therapistString = therapist.reply(patientString);
            System.out.println("\n" + therapistString);
         }
      }
   }
```

Implementation of the `Therapist` Class

```
/* Case Study 14.2: Therapist.java
1) This class emulates a nondirective psychotherapist.
2) The major method, reply, accepts user statements and generates
   a nondirective reply.
*/

import java.util.*;

public class Therapist{

   private Set<String> hedgeSet;                // The set of hedges
   private Set<String>  qualifierSet;           // The set of qualifiers
   private Map<String, String>   replacementMap;// The map of
                                                // replacement words
   public Therapist(){
      hedgeSet = new HashSet<String>();
      hedgeSet.add("Please tell me more");
      hedgeSet.add("Many of my patients tell me the same thing");
      hedgeSet.add("It is getting late, maybe we had better quit");
      qualifierSet = new HashSet<String>();
      qualifierSet.add("Why do you say that ");
      qualifierSet.add("You seem to think that ");
      qualifierSet.add("So, you are concerned that ");
      replacementMap = new HashMap<String, String>();
      replacementMap.put("i", "you");
      replacementMap.put("me", "you");
      replacementMap.put("my", "your");
      replacementMap.put("am", "are");
   }

   public String reply(String patientString){
   // Replies to the patient's statement with either a hedge or
   // a string consisting of a qualifier concatenated to
   // a transformed version of the patient's statement.
   //  Preconditions  -- none
   //  Postconditions -- returns a reply
      String reply = "";              // The therapist's reply
      int choice = randomInt(1, 3);   // Generate a random number
                                      // between 1 and 3
      // If the patient says nothing, then encourage him.
      if (patientString.trim().equals(""))
         return "Take your time. Some things are difficult to talk about.";
      // Else reply with a hedge or a qualified response
      if(choice == 1)
```

```
      reply = hedge(hedgeSet);           // Hedge 1/3 of the time
   else if (choice == 2 || choice == 3)
      reply = qualifier(qualifierSet) +    // Build a qualified response
              changePerson(patientString);// 2/3 of the time
   return reply;
}

private String hedge(Set<String> hedgeSet){
// Selects a hedge at random
//  Preconditions -- the hedge set has been initialized
//  Postconditions -- returns a randomly selected hedge
   return selectRandom(hedgeSet);
}

private String qualifier(Set<String> qualifierSet){
// Selects a qualifier at random
//  Preconditions  -- the qualifier set has been initialized
//  Postconditions -- returns a randomly selected qualifier
   return selectRandom(qualifierSet);
}

private String changePerson(String str){
// Returns a string created by swapping i, me, etc. for you, your, etc.
// in the string str
//  Preconditions  -- none
//  Postconditions -- returns the created string
// Tokenize str
   Scanner tokens = new Scanner(str);
   String result = "";                    // Create a response string
   // Build the response from replacements of the tokens
   while (tokens.hasNext()){
      String keyWord = tokens.next();
      String replacement = findReplacement(keyWord);
      result = result + replacement + " ";
   }
   return result;
}

private String findReplacement(String keyWord){
// Returns the value associated with the keyword or the keyword itself
// if the keyword is not in the map.
//  Preconditions  -- the replacement map has been initialized
//  Postconditions -- returns the replacement
   keyWord = keyWord.toLowerCase();
   if (replacementMap.containsKey(keyWord))
      return (String) replacementMap.get(keyWord);
   else
      return keyWord;
}

private String selectRandom(Set<String> set){
// Selects an entry at random from the set
//  Preconditions  -- the set is not empty
//  Postconditions -- returns the random entry
   int index = randomInt(0, set.size() - 1);
```

```
        Iterator<String> iter = set.iterator();
        for (int i = 0; i < index; i++)
            iter.next();
        return iter.next();
    }

    private int randomInt(int low, int high){
    // Generate a random number between low and high
    //  Preconditions  -- low <= high
    //  Postconditions -- returns the random number
        return (int) (low + Math.random() * (high - low  + 1));
    }
}
```

14.8 The Glue That Holds Collections Together

We close this chapter as we began it, by asking about the ways in which collections are similar. All collections contain objects, have a definite size at any given time, and can grow or shrink in size. Are there any operations that all collections have in common? These operations, if they exist, would serve as a kind of glue that holds collections together, allowing them to interact in very powerful ways.

Addition, Removal, and Membership

Each type of collection we have examined thus far supports some form of addition and removal of elements. Some collections have more than one way to do this; for example, lists have index-based and object-based versions of additions and removals, and position-based versions via an iterator. It seems that every collection allows at least object-based additions and removals. Indeed, the object-based methods `add` and `remove` for lists and sets have the same signature. These collections also support a method to test an object for membership in the collection, namely, `contains`. Maps include variants of these operations for keys and values. Clearly, any list of general collection operations will have some of these basic operations among them or will use them to implement others.

Iterators

Earlier in this chapter, we showed how to access items in lists and sets with an iterator. The `Map` method `keySet`, which returns a set of a map's keys, allows you to use an iterator on this set to traverse the map. It is a critical fact that iterators work the same way, whether they are used with lists, sets, or maps. Thus, iterators belong in the list of common collection operations.

Other Operations

One common operation on any collection removes all its elements, thus producing an empty collection. The implementation code for this method, called `clear`, can be the same for any collection if we use an iterator that supports the method `remove`. The code for `clear` simply iterates through the elements and uses the `Iterator` method `remove` on each element.

Programmers often must construct operations that involve more than one collection. We saw examples earlier in this chapter of the set operations union, intersection, and difference, which the `Set` collection does not include but which we can implement quite easily. Other, more general tasks that apply to any pair of collections cry out for standard methods. Consider the following four:

- Add all the elements of one collection to another collection.
- Remove all the elements of one collection from another collection.

- Retain only the elements of one collection that are also present in another collection.
- Determine whether or not one collection contains all the elements in another collection.

It would be quite simple to implement all of these operations for any pair of collections, including two collections of different types, with just an iterator and the methods `add`, `remove`, and `contains`.

The `java.util.Collection` Interface

Several of the common collection operations discussed thus far are included in the `java.util.Collection` interface. Both the `List` and the `Set` interfaces extend this interface, so these operations are available to lists and sets. Table 14-16 describes the `Collection` methods.

TABLE 14-16
The `Collection` interface

METHOD	WHAT IT DOES
`boolean isEmpty()`	Returns `true` if the collection contains no elements, or `false` otherwise.
`int size()`	Returns the number of elements currently in the collection.
`void clear()`	Makes the collection be empty.
`boolean contains(Object element)`	Returns `true` if the collection contains `element`, or `false` otherwise.
`boolean remove(Object element)`	Removes `element` if it exists and returns `true`, or returns `false` if `element` does not exist.
`boolean add(Object element)`	Adds `element` to the collection if `element` does not already exist and returns `true`. Returns `false` otherwise.
`Iterator<E> iterator()`	Returns an iterator on the collection, allowing the programmer to traverse it with an enhanced `for` loop.
`boolean containsAll(Collection<?> c)`	Returns `true` if all of the elements in `c` are also contained in the receiver collection, or `false` otherwise.
`boolean addAll(Collection<? extends E> c)`	Adds all of the elements in `c` to the receiver collection and returns `true`, or `false` otherwise.
`boolean removeAll(Collection<?> c)`	Removes all of the elements from the receiver collection that are also in `c`. Returns `true` if at least one element was removed, or `false` otherwise.
`boolean retainAll(Collection<?> c)`	Retains all of the elements from the receiver collection that are also in `c` and removes the rest. Returns `true` if at least one element was removed, or `false` otherwise.

Some of the operations in Table 14-16, such as `removeAll`, are listed as optional. This means that the implementing classes must include these methods but need not support them. In practical terms, if a given method is not supported, the calls to this method in a client's code will compile but these calls will throw an `UnsupportedOperationException` at run time.

Several operations expect a parameter of type `Collection`. This means that you can pass as a parameter an object of any class that implements the `Collection` interface. That would be all of the list and set classes in `java.util`, as well as any programmer-defined classes that implement `Collection`.

Finally, the list and set classes include constructors that expect a parameter of type `Collection`. These constructors provide a simple way of converting one type of collection to another.

Some Examples of the Use of General Collection Operations

The following code segments show some examples of the power of the collection methods.

Set union

```
Set<String> unionSet = new HashSet<String>(set1).addAll(set2);
```

Set intersection

```
Set<String> intersectionSet = new HashSet<String>(set1).retainAll(set2);
```

Set difference

```
Set<String> differenceSet = new HashSet<String>(set1).removeAll(set2);
```

Sort a list, removing duplicates

```
List<String> sortedList = new ArrayList<String>(new TreeSet<String>
  (unsortedList));
```

The `Collections` Class

Given the wide variety of operations on collections, the programmer can write new methods for almost any occasion. As you saw in Chapter 12, you can write methods for searching and sorting arrays using several different algorithms. Other useful methods, such as searching for a minimum or maximum value, also can be written easily. Several of these operations are so common that `java.util` implements them in a set of `static` methods, primarily for use with lists. These methods are included in the `Collections` class. Note that `Collections` (in the plural) is the name of a *class*, like the `Math` class, whereas `Collection` (in the singular) is the name of an *interface* that is implemented by a set of collection classes.

Using the `Collections` class, you can sort and search a list of strings as follows:

```
Collections.sort(list);                 // Assume list contains only strings
Collections.binarySearch(list, "Mary");// See if "Mary" is in list
```

Table 14-17 describes some commonly used `Collections` methods, followed by an explanation of the extraordinary syntax.

TABLE 14-17
Some `Collections` methods

METHOD	WHAT IT DOES
`static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)`	Searches the specified list for the specified key using the binary search algorithm
`int static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll))`	Returns the maximum element of the given collection, according to the natural ordering of its elements
`int static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> coll))`	Returns the minimum element of the given collection, according to the natural ordering of its elements
`static void reverse(List<?> list)`	Reverses the order of the elements in the specified list; this method runs in linear time
`static void shuffle(List<?> list)`	Randomly permutes the specified list using a default source of randomness
`static <T extends Comparable<? super T>> void sort(List<T> list)`	Sorts the specified list into ascending order, according to the natural ordering of its elements

Each of the method signatures starts with the modifier `static`. This, of course, means that the user of the method sends a message to the `Collections` class, not to an instance of any class. The method headings for `shuffle` and `reverse` have the simplest syntax. They allow `List` parameters of any element type, as specified by the wildcard symbol `?`. In the case of `shuffle`, this is actually shorthand for the syntax `static <T> void shuffle(List<T>)`.

The syntax becomes more alarming in the headings of the other methods. Let's start with `sort`. This method expects a `List` with an element type parameter `T`. However, the method works not just for any element type `T`, but for an element type `T` that is restricted by implementing the `Comparable` interface. This restriction is expressed not in the formal parameter, but in the type parameter information between the modifier `static` and the return type `void`. Moreover, we find there not just the familiar `<T extends Comparable<T>>` but instead the more daunting `<T extends Comparable<? super T>>`. This means not just a type `T` that implements `Comparable` but also any *supertype* of `T` that implements `Comparable`. For example, we might have a list of bank accounts, wherein there are not only checking accounts but also savings accounts. If the two specific types of accounts are subclasses of a common bank account class that implements `Comparable`, then we can apply this sort method to that list.

The notation used with the methods `min` and `max` allows you to use these methods with collections of any objects that extend type `T` and also implement the generic `Comparable` interface.

Fortunately, you don't have to memorize this syntax or even thoroughly understand it to use these methods!

SUMMARY

In this chapter, you learned:

- Collections are container objects that organize a number of data elements in a program.
- Collections typically grow or shrink in size with the needs of a program.
- Lists, stack, queues, sets, and maps are types of collections that support special-purpose access to their elements.
- A list is a linear sequence of elements that are accessed by position.
- A stack is a linear sequence of elements that are accessed at one end, called the top.
- A queue is a linear sequence of elements that are added at one end and removed at the other end.
- A set is an unordered collection of unique elements.
- Sorted sets and maps allow programmers to visit their elements in sorted order.
- An iterator is an object that allows the programmer to traverse the elements in a collection.
- The logical behavior of a collection is specified in its interface.
- Programmers can choose among different classes that implement the same collection interface. This choice depends on the run-time performance of the implementing class.

VOCABULARY *Review*

Define the following terms:

association list	keyed list	stack
collection	list	table
dictionary	map	type parameter
hashing	queue	type variable
iterator	set	wrapper class

REVIEW *Questions*

FILL IN THE BLANK

Complete the following sentences by writing the correct word or words in the blanks provided.

1. Two examples of collections whose elements are ordered by numeric index position are _____ and _____.
2. A collection that supports access to elements in Last In First Out order is a(n) _____.

3. A collection that supports access to elements in First In First Out order is a(n) _____.
4. A(n) _____ is a collection that contains unique elements in no particular order.
5. _____ and _____ are two classes that implement lists in Java.
6. The _____ is a Java list class that supports access to elements in constant time.
7. A(n) _____ is a collection that associates each value with a unique key.
8. A(n) _____ is an object that allows a programmer to visit each element in most collections.
9. _____ is the name of the Java interface that all list classes implement.
10. _____ is the name of the Java interface that both lists and sets implement.

PROJECTS

PROJECT 14-1

Write a program that prompts the user for an input text filename and an output text filename. The program reads lines of text from the file, adds these lines to a list of strings, converts the strings in the list to uppercase, and then writes the strings to the output file.

PROJECT 14-2

A data collection program has saved integers in a text file. The order of the integers in the file is not important, and the user of this data is only interested in the unique integers. Write a program that prompts the user for an input text filename, inputs the integers from that file, and prints the unique integers in this file to the terminal screen. The program should use a Java set to solve this problem.

PROJECT 14-3

Write a program that evaluates postfix expressions. This program prompts the user for an input string. The input string should contain a syntactically correct postfix expression, in which the operands and operators are separated by at least one space. The program uses the stack-based algorithm discussed in this chapter to evaluate the input expression and print its value. In addition to a stack, you should use two `Scanner` objects to solve this problem. One scanner obtains the user's input string. The other scanner extracts the operators and operands, also known as tokens, from this string.

PROJECT 14-4

A concordance algorithm determines the unique words in a text file and their frequencies. Write a program that uses a map to implement this algorithm. The program should read words from the file and add them to the map if they are not already in it. When a word is first added to the map, the initial frequency value of 1 is associated with the word as a key. When a word is already a key in the map, the associated frequency value is just incremented by one. The program should traverse the map to print its contents when the file input process is finished.

PROJECT 14-5

The following paragraph describes a simplified version of the card game of War. For a more complex version, see Hoyle's *Rules of Games* (New York: Signet Books, 2001).

There are two players in the game of War. During the course of a game, each player will have three piles of cards, named an unplayed pile, a war pile, and a winnings pile, respectively. The game moves forward as cards move from the unplayed piles to the war piles and then to the winnings piles. The game ends when a player's unplayed pile has no more cards. At that point, the player with the largest winnings pile wins the game. Here are the detailed rules for moving cards:

1. Each player is dealt 26 cards to her unplayed pile.
2. Repeat Steps 3 through 5 until one or both unplayed piles become empty.
3. Each player plays the topmost card from his unplayed pile by placing it face up on his war pile.
4. If the cards have the same rank, repeat Step 3.
5. Otherwise, move both war piles to the winnings pile of the player who has the card of a higher rank at the top of her war pile.
6. The player with the largest winnings pile wins.

Write a terminal-based program that uses the resources developed in Case Study 1 to play this game. The computer should make all of the moves for both players and display the cards played on each move. You should use the appropriate objects to represent the piles of cards.

PROJECT 14-6

Convert the terminal-based program of Project 14-5 to a GUI program. The main window should contain a labeled text field for each player and a single Play button. The cards are dealt at start-up, but play does not start until the user clicks Play. The text fields, initially empty, are updated to display the information on the cards played from each hand after each click on Play. When a game ends, a message dialog box pops up to announce the winner. A Reset button allows the user to start a new game at any time.

PROJECT 14-7

After working on Project 14-6, Jack realizes that the game would be a lot more fun if he displayed realistic images of cards. In this project, modify the `Deck` and `Card` classes so that they are capable of displaying images. Write a short GUI tester program that cycles through a deck of cards and displays their images in a single panel.

Free images for decks of cards can be found at many Web sites. A directory containing some images is available from your instructor. Take care not to use a copyrighted image and make sure you have permission to download any images from the Web.

Each card should have an additional instance variable of type `ImageIcon` to hold its image. A new constructor accepts an image's filename as a parameter. The constructor loads the image from the file and sets the card's image variable. The `Card` class also includes a public method `getImage` that return's a card's image.

The `Card` class should have an additional instance variable of type `ImageIcon` to hold its image. A new `Card` constructor accepts an image's filename as a parameter. The constructor loads the image from the file and sets the card's image variable. The `Card` class also includes a public method `getImage` that returns a card's image.

The `Deck` class should have a new constructor that accepts a directory path name as a parameter. When the constructor enters the loop to create new cards, it uses the pathname to build the appropriate filename of each card's image file. You should adopt a scheme for naming these files that supports this process, such as "s1.gif" for the Ace of Spades' file, and so forth. Resulting filenames are then passed to the new `Card` constructor mentioned earlier.

The tester program creates a new deck at start-up. Its window contains a single panel and two buttons, labeled Deal and Reset. Each time that Deal is clicked, a new card is dealt from the deck and the card's image is displayed in the panel. When no more cards are available, a message dialog box pops up and the panel is cleared. Reset clears the panel and creates a new deck of cards at any time. The window's size should be large enough to accommodate a card's image.

The panel is a specialized version of the `ColorPanel` class used earlier. A new instance variable of type `ImageIcon` is initially `null`. A new public method `setImage(anImageIcon)` sets this variable and repaints the panel. When the panel's image variable is `null`, `paintComponent` just calls itself in the superclass, effectively clearing the panel. Otherwise, `paintComponent` also paints the image.

The window class calls `setImage` with a card's image when it should be displayed in the panel and calls the same method with `null` to clear the panel.

PROJECT 14-8

Use Jack's modified resources from Project 14-7 to develop a GUI-based game of War. (*Hint*: Replace the text fields with card panels.)

PROJECT 14-9

Participants in a conversation sometimes refer to earlier topics in the conversation. Add this capability to the therapist of this chapter's Case Study 2. To implement this capability, add a history list containing the patient's input sentences to the program. At random intervals, the program should select a sentence from the history list, transform the persons in it, and return it as its reply. The program should add a patient input sentence to this list only after a reply to that sentence is computed. Also, be sure that the list contains no fewer than three sentences before you access it.

CRITICAL *Thinking*

Jill is trying to decide whether to use an array or an instance of Java's `ArrayList` class in an application. Explain to her the costs and benefits of using one data structure or the other.