[CHAPTER] 9

The Java Collection Framework

9.1 Introduction

In the early days of software development, programmers who wanted to produce a sequence of random numbers had to do all the work themselves. They had to study the algorithms for generating random values, write and debug the code, and test the statistical properties of the random numbers they produced. Today, that approach would be utterly antiquated. Software developers now go to a program library, locate the routines they need, import them into their code, and use them. (In Java, the code to produce random numbers is in the `Random` class, in the package `java.util`.) Similarly, if programmers need to sort an array of numerical values today, it is highly unusual for them to write their own Quicksort, merge sort, or heap sort method. Instead, they use one of the many sorting routines provided by Java. (The `Arrays` class in the `java.util` package includes an implementation of merge sort that sorts information stored in an array.)

The creators of these routines realized that one of the best ways to increase software productivity is to provide libraries of carefully tested, highly optimized routines for everyday tasks. By using these libraries, programmers would not have to write every single function themselves, but instead could think and work at a higher level of abstraction. Rather than building a program from scratch, they could use components from a class library, reducing development time and producing programs that are more likely to be bug-free.

One of Java's more useful features is the Java Collection Framework. This set of classes provides many of the data structures discussed in the last three chapters (for example, lists, stacks, sets, and maps), so programmers no longer need to build these structures from scratch. Using the Java Collection Framework, software developers can increase productivity and significantly reduce implementation time and costs.

The existence of these data structure libraries does not mean that the concepts and ideas of the previous three chapters are no longer relevant. Software designers must have a solid grounding in the fundamental characteristics of data structures and must be able to determine which data structures work best in a given situation. The ability to build these structures is less important than being able to select the best data structure to solve a particular task. For example, how would you know whether an array or a linked list implementation of a queue is best for a given situation? How do you evaluate the time-space trade-offs between an adjacency matrix and an adjacency list representation of a graph? What are the efficiency implications of using a binary search tree in place of an unsorted array?

You can only answer these questions if you thoroughly understand the basic characteristics of data structures and their associated algorithms. However, because of libraries such as the Java Collection Framework, your primary focus has changed from *implementing* and *building* data structures to *understanding*, *analyzing*, and *using* them. This knowledge allows software designers to make intelligent, well-informed decisions about which classes in the framework are best for a given application.

This chapter describes the services and capabilities of the Java Collection Framework. It relies on the data structure concepts in Chapters 6, 7, and 8, makes extensive use of the algorithm analysis in Chapter 5, and makes heavy use of generics, which were introduced in Chapters 2 and 3.

9.2 The Java Collection Framework

9.2.1 Overview

A **framework** is a unified architecture for providing a set of related services—a collection of routines designed to work together seamlessly and to provide a common external "look and feel" to its users. The **Java Collection Framework**, which is found in the package `java.util`, represents a unified architecture for creating and manipulating important and widely used data structures.

Java is not the only programming language to provide this type of framework. For example, in 1994, both the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO) voted to make the **Standard Template Library (STL)** an official part of the C++ language. Like the Java Collection Framework, the STL provides such common data structures as lists, stacks, queues, and sets, as well as the algorithms required to manipulate them. However, many of these earlier collection frameworks were extremely complex and difficult to learn. Programmers often found it easier to design and build their own code from scratch rather than spend the time learning how to make effective use of existing collection classes. To rectify this problem, the Java Collection Framework was carefully designed to be simple and straightforward to use. By the end of this chapter, you should be able to exploit this package of data structure and algorithmic services.

In addition to increasing productivity, the Java Collection Framework can improve software development in several other ways:

- *Better run-time efficiency*—The routines in the collection framework are carefully optimized to provide the best possible run-time performance. It would be time consuming and technically demanding for individual programmers to duplicate these optimizations every time they designed and built their own methods.
- *Increased program generality*—Programmers can use standardized interfaces to represent a collection and are free to change the underlying implementation. For example, we can decide to implement a list in our program using either an `ArrayList` or a `LinkedList`, depending on the problem. We can even change the implementation without affecting the correctness of the program.
- *Interoperability*—Unrelated programs can more easily exchange collections with each other using the standardized services of the Java Collection Framework and thus facilitate sharing information across a network.

The following sections explain the structure and design of the Java Collection Framework so that you can begin to take advantage of its many capabilities.

REUSE IT OR LOSE IT

NASA has announced plans for the next generation of rockets that will carry astronauts to the moon and beyond. Many people who have looked at the designs remark how similar they are to those of the Apollo and shuttle programs. In fact, NASA makes the following statement on one of its Web sites: "This journey begins soon, with development of a new spaceship. Building on the best of Apollo and shuttle technology, NASA's creating a 21st century exploration system that will be affordable, reliable, versatile, and safe."

Some people criticized NASA for not inventing new technologies, but it was simply doing what successful engineers have done for years: reusing components that have been tested and proven safe and efficient. Think about it. If you were an astronaut preparing to go into orbit, would you rather be climbing into a brand-new spaceship whose parts are being used for the first time? Or, would you prefer to know that the spaceship was designed and constructed with parts that have been successfully used? Reusability is a fundamental principle of engineering.

continued

The design and use of reusable software components has been a goal of programmers for many years. The traditional form of reuse is a library that consists of prewritten usable components; extensive libraries are now commonly included with most modern programming languages. This text focuses on Java, but languages such as C++, C#, and Ruby also come with extensive libraries. The open source movement has introduced a new form of reuse; you can reuse an entire design or subsystem instead of an individual component.

Reuse not only helps programmers to develop correct and efficient software, it decreases development time, which in turn reduces the amount of time needed to bring a software product to market and reduces maintenance costs. To maintain your proficiency in today's quickly evolving development environment, you must learn to reuse code.

9.2.2 Collections

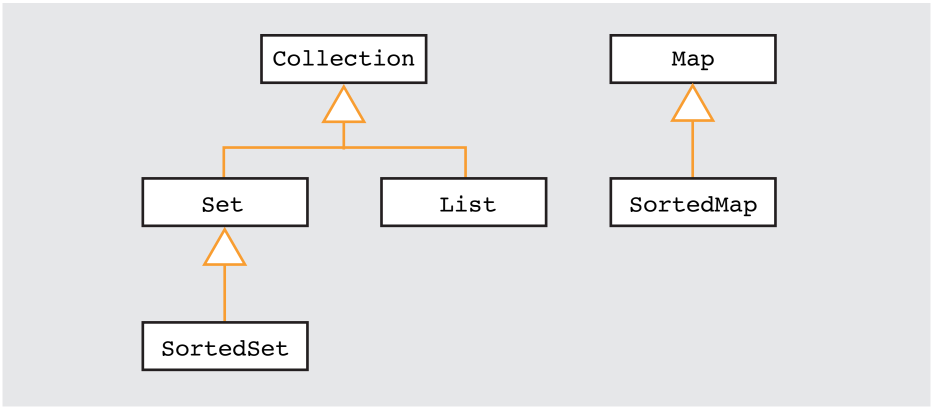
The Java Collection Framework is organized around the concept of a collection. A **collection** is an object that groups together multiple elements into a single entity. In Chapter 6, we referred to this idea as a *composite data type* or a *data structure*. Collections store, retrieve, and manipulate information, and transmit it from one application to another (for example, sending a list of items from a client process to a server). Collections typically represent data items that form a naturally related group of entities, such as a telephone directory, a deck of cards, a list of products purchased by a user, or the set of integers in the range [0...10].

The Java Collection Framework is made up of three components: interfaces, implementations, and algorithms. The interfaces of the Java Collection Framework, like the interfaces that were introduced in previous chapters, provide the specifications (or behaviors) for a particular type of collection. Each interface lists the set of services the collection can provide, independent of the implementation that ultimately provides these services. The implementations represent the specific way we construct the methods in an interface. Some interfaces may require multiple implementations because of differences in their performance characteristics for certain types of operations. We did this in previous chapters, in which we often described multiple implementations (for example, array-based and linked list) for a single interface. One implementation might perform well with retrievals, while another does poorly with retrievals but is much faster with insertions and deletions. Finally, the algorithms are methods that manipulate the data stored in a collection. For example, the Java Collection Framework includes algorithms that can sort a collection, search a collection, or randomize the order in which elements appear in a collection. These algorithms are chosen to provide their services in the most efficient method possible.

We describe each of these three components in the following sections.

9.3 Interfaces

The six interfaces in the Java Collection Framework are shown in the UML class diagram of Figure 9-1.



[FIGURE 9-1] Interfaces in the Java Collection Framework

9.3.1 The `Collection` Interface

9.3.1.1 Maintaining a Collection

The `Collection` interface is the root of the collection hierarchy, and represents the most general and flexible of all collection types. For example, some types of collections allow duplicates, but others do not; some collections represent a position-dependent sequence of elements, but others are unordered. Basically, the `Collection` interface represents a group of elements with virtually any characteristics.

The Java Collection Framework does not provide an implementation for the `Collection` interface. Instead, you take this highly general interface and create a number of lower-level "subinterfaces" that impose specific characteristics on a collection. The main use of the `Collection` interface is to describe a general set of methods that are applicable to all collection types, because every subinterface of `Collection` inherits all these methods. Another use of the `Collection` interface is to pass collections of objects between applications with the maximum amount of generality. The `Collection` interface, found in the Java package `java.util`, is shown in Figure 9-2.

```
public interface Collection<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);                   // Optional
    boolean remove(Object element);           // Optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); // Optional
    boolean removeAll(Collection<?> c);        // Optional
    boolean retainAll(Collection<?> c);        // Optional
    void clear();                              // Optional

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

[FIGURE 9-2] `Collection` interface

You should recognize the methods in the `Collection` interface based on what you have learned in the previous three chapters. The generality of the `Collection` interface means that the semantics of its methods are defined in the most general way possible. For example, the rules for the behavior of the `add()` method say that the object is added to the collection in the "proper way." This means you cannot add the object to the collection if it is there, and duplicates are not allowed. If duplicates are permitted, the method adds the object to the collection. The rules for the behavior of `remove()` are specified in a similar fashion, in that it is guaranteed to work consistently with the characteristics of the specific collection. For example, if duplicates are allowed, then `remove()` takes out a *single* instance of the object.

The `contains()` method allows you to determine if a given object is a member of a collection or not. The method uses the `equals()` method to determine if two objects are equivalent. The code that implements the `contains()`method consists of a loop that steps through each element in the collection and invokes the `equals()` method on the current object to determine if it is equivalent to the object in question. Whenever you must determine if two objects are equivalent in the collection framework, the `equals()` method makes the determination. When writing classes whose objects will be placed in a collection, the class must include an `equals()` method appropriate for comparing objects that belong to the class.

The four methods whose names end in *All* in Figure 9-2 are called **bulk operations**, and perform an operation on the entire collection in a single step. The method `s1.containsAll(s2)` returns true if every element in collection `s2` is also in `s1`. The operation `s1.addAll(s2)` adds every element in collection `s2` to `s1`. This addition is consistent with the properties of collection `s1`. The method `s1.retainAll(s2)` keeps an element in `s1` only if it is also present in collection `s2`.

Finally, `s1.removeAll(s2)` removes from `s1` all elements that are in collection `s2`. The removal is consistent with the properties of collection `s1`. The method `clear()` removes all elements from the collection. Finally, the `toArray()` methods allow the elements in a `Collection` to be stored as an array. The placement order of elements in the array must be consistent with the specific characteristics of the collection.

If you examine the method signatures in the `Collection` interface, you might wonder why some methods, such as `add()` and `addAll()`, use generic types, while others, such as `remove()` and `removeAll()`, do not. The designers of the Java Collection Framework wanted you to be able to create collections that can contain any sort of object, while ensuring that any operation you perform on the collection can be verified as safe by the compiler. For a collection, this means we must ensure that all of its elements and objects are type compatible (in other words, they all share a common set of behaviors). For example, if you have a collection of strings, you want to make sure that you cannot inadvertently add numbers to the collection.

This is why the `add()` and `addAll()` methods specify the type of the elements you can add to the collection. Once you specify the type of the objects in the collections, the compiler only accepts invocations of the `add()` method whose arguments either match the type of the collection or extend it in some way. Likewise, the `addAll()` method accepts arbitrary collections as an argument; however, the collection must contain objects that either match or extend the type of the added objects. On the other hand, when specifying elements to remove from a collection, we only need to ensure that we can invoke `equals()` on the elements we want to remove and the elements in the collection. Because `equals()` is defined to work for the class `Object`, you do not need to specify the element types in methods such as `remove()` and `removeAll()`.

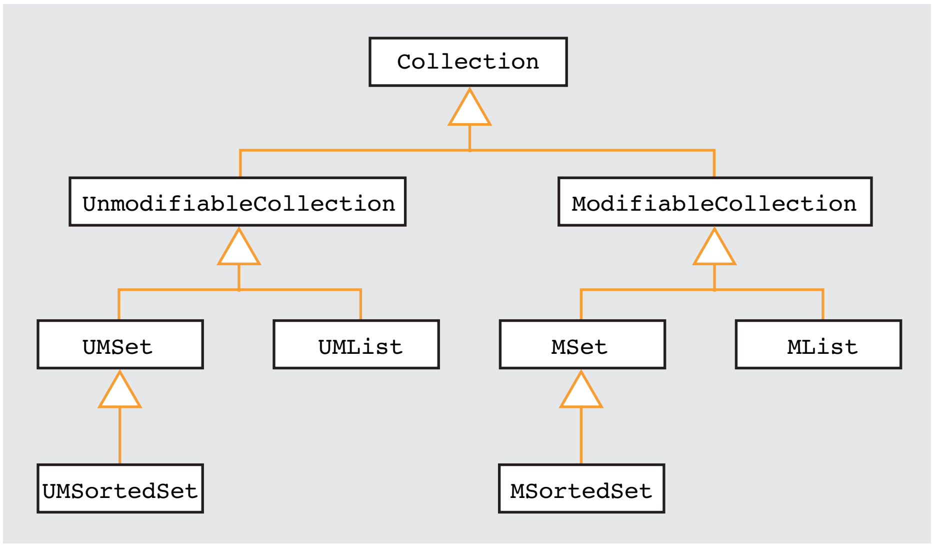
The interface in Figure 9-2 includes 13 methods, but six of these are optional. If a class that implements `Collection` does not provide an implementation for one of these optional methods, it throws an `UnsupportedOperationException`. (You will learn how to deal with exceptions in Chapter 10. For now, you can view an exception as an error.)

The designers of the Java Collection Framework made the controversial decision to include optional methods in interfaces. It was controversial because a user of a class that implements the `Collection` interface cannot be sure that a call to one of these optional methods will not "break" the program. For example, this harmless-looking sequence:

```
Collection<String> c;            // c is a collection
boolean ok = c.add( "Grumpy" ); // try to add a string to c
```

could actually cause this program to terminate with an exception because `add()` is an optional method, and it may not have been implemented by `c`.

This approach reduces the number of interfaces that need to be included in the framework. For example, the Java Collection Framework supports the idea of a read-only collection—a collection whose contents can be accessed but not changed. Clearly, this type of collection should not contain methods that change the contents of the collection, such as `add()` or `remove()`. It would mean that the top-level interface could not include these methods, resulting in an inheritance hierarchy that might resemble the one in Figure 9-3.



[FIGURE 9-3] The effect of adding interfaces to the collection framework

By adding a separate interface for read-only collections, called `UnmodifiableCollection`, we would approximately double the total number of interfaces in the framework. Now consider the fact that the framework also provides support for synchronized collections to enable concurrent programming. This means that the number of interfaces in the framework would more than triple in size, as it would now include `UnmodifiableSynchronizedSet`, `UnmodifiableSet`, `ModifiableSynchronizedSet`, and so on. Therefore, a design decision was made to reduce the number of interfaces in the collection by marking some methods as optional. Note that, even though a method such as `add()` might be optional, it must still be included in any class that implements the interface. In the sense of the collection framework, *optional* means that a method may throw an exception if it is invoked. For example, a read-only collection must still provide an `add()` method, but that method does not actually add an element to the collection. Instead, it throws an exception whenever it is invoked. Our examples assume that all optional methods have been fully implemented.

In our discussion so far, we have implied that a collection *holds* elements. In reality it does not, but instead stores a reference to each object that is in the collection. This is an important distinction, because it means that you can only put objects, not the values of primitive types, in a collection. So, it is perfectly legal to create a collection that can hold instances of the `String` class, but you cannot create a collection that holds `int` values such as 1, 52, or 34.

This seems like a severe restriction that limits the use of collections in your program. Recall from Chapter 3 that Java provides wrapper classes for each of the primitive types. The class `Integer`, contained in the `java.lang` package, is the wrapper class for `int` values. An instance of the `Integer` class has as its state a single integer value. The following code shows how to create a `Collection` that contains `Integer` values.

```
// Create a collection of integers
Collection<Integer> c = new ArrayList<Integer>;
// Add some numbers to the collection
for ( int i = 0; i < 10; i++ ) {
    c.add( new Integer( i ) );
}
```

We also discussed autoboxing and autounboxing in Chapter 3. This feature of Java makes code less tedious to write. Because a collection can only hold references to objects, the compiler automatically boxes any primitive values using the appropriate wrapper class, and unboxes any primitive values when they are taken out of a collection. Therefore, in a

Java program you can effectively use `int` values and `Integer` objects interchangeably. We rewrote the previous code example to take advantage of autoboxing:

```
// Create a collection of integers
Collection<Integer> c = new ArrayList<Integer>;
// Add some numbers to the collection
for ( int i = 0; i < 10; i++ ) {
    c.add( i );  // i is autoboxed to new integer( i )
}
```

The autoboxing feature makes the code much easier to read. Keep in mind that, behind the scenes, the compiler is actually creating instances of the appropriate wrapper class and adding a reference to that object to the collection. So, although it may appear that the collection is holding `int` values, it really holds references to instances of the `Integer` class. We did not explicitly create the `Integer` objects; the compiler created them automatically.

9.3.1.2 Iterators

An **iterator** is an object that allows us to move in order through a collection from one element to another. The order in which we sequence through a collection depends on the type of collection we are using. For example, in a linear structure such as a list, we move from a node to its unique successor. In a tree or a graph, we have more flexibility for iterating through the collection (e.g., preorder, postorder, breadth-first, depth-first). In a set, it does not matter how we do the iteration because its elements are position independent.

In the previous three chapters, all the data structures used an **internal iterator**—rather than being an object separate from the collection itself, the iterator was a state variable of the collection, and was part of the collection. For example, the `LinkedList` class in Figure 6-16 included a state variable called `cursor`, which identified a position within the list. We iterated through a list using the instance methods `first()`, `next()`, and `isOnList()`:

```
LinkedList<String> aList = new LinkedList<String>();

// Set the cursor to the beginning of the list
aList.first();

while ( aList.isOnList() ) {
    // Perform some operation on the current node
    aList.next(); // Move cursor to the next node
}
```

In this example, `cursor` is an internal iterator because it is part of the state of the `LinkedList` object `aList`. However, the problem with this technique is that we can only use the number of iterators directly provided by the collection. For example, because the `List` class of Chapter 6 contains only a single cursor, we could not perform two iterations over the list at the same time.

To eliminate this restriction, the Java Collection Framework uses **external iterators**. An external iterator is an object that references an element in a collection but is separate from the collection object itself. The `Iterator` interface is shown in Figure 9-4.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); // Optional
}
```

[FIGURE 9-4] `Iterator` interface

An object that implements the `Iterator` interface generates a sequence of elements from the collection, one at a time. The method `next()` returns the "next" object in this collection, using ordering rules that are appropriate for the particular type of collection it is using. In addition to returning a reference to the current object in the iteration, invoking `next()` also causes the cursor to advance to the next element in the iteration. The method `hasNext()` returns true if the iteration contains more elements and false otherwise.

One of the most common uses of an iterator is to examine every element in a collection. For example, if `aCollection` is an instance of a class that implements the `Collection` interface that holds `String` objects, then we can use an `Iterator` to print the elements in the collection as follows:

```
// Create an iterator over the collection
Iterator<String> i = aCollection.iterator();

// Iterate through the collection, one element at a time
while ( i.hasNext() ) {
    // Print the next element in the iteration and
    // advance the cursor
    System.out.println( i.next() );
}
```

You can use a for loop to iterate over collections, as shown in the following example:

```
// Iterate through the collection, one element at a time
for ( String s : aCollection ) {
    // Print the next element in the iteration and
    // advance the cursor
    System.out.println( s );
}
```

The for loop in the preceding code executes the body of the loop *for each* string in the collection `aCollection`. This form of the loop is often called a **for-each** loop instead of a for loop. The general form of the for-each loop is shown in the following code:

```
for ( type var : collection ) {
     loop body
}
```

In this example, *type* is the type of the items in the collection you are iterating over, *var* is the loop control variable that refers to the current element in the iteration, and *collection* is the collection you are iterating over. Recall from Chapter 3 that you also can use a for-each loop to iterate over arrays. In fact, you can use the for-each loop to iterate over any object that implements the `Iterable` interface.

From a functional standpoint, a for-each loop is identical to code that obtains a reference to an iterator, and uses that reference in conjunction with a while loop to iterate over a collection. In fact, if you look at the code the compiler generates for a for-each loop, you can see that it is identical to the code that uses the while loop. The advantage of the for-each loop is that you do not have to explicitly obtain a reference to the iterator that accesses the elements in the collection. When using a for-each loop, you are not aware that an iterator is being used; in fact, it is not even possible for you to obtain a reference to the iterator. Using a for-each loop makes your code easier to read and less tedious to write.

The `Iterator` interface includes an optional `remove()` method that removes the last element returned by a call to `next()` from the collection. Because `remove()` operates on the element that was already returned by the last call to `next()`, the removal of an object from the collection does not affect the state of the iteration, as the cursor has already advanced past the element that was removed. The `remove()` method can only be called once per call to `next()`.

Note that because you cannot access the iterator in a for-each loop, you cannot use the iterator's `remove()` method to remove elements from a collection within a for-each loop. A for-each loop works fine when you want to do a simple iteration over a collection, but if you need to invoke any of the iterator's methods directly, you cannot use the for-each loop.

A helpful feature of external iterators is that we can create and use as many over a collection as we want. We are no longer limited to the number of internal iterators provided by the collection object itself. Because the state of an iterator is separate from the state of the collection over which it iterates, and because you can have multiple iterators for the same collection, any changes you make to the collection while there is at least one active iterator over it (in other words, one whose `hasNext()` method returns true) may result in unpredictable behavior. For example, consider the situation in which there is an active iterator over a list, and the iterator is currently viewing the first element. If `remove(0)` was invoked on the list, how would the iterator know that the element being viewed has been deleted from the list?

Internally, any object in Java that can have an external iterator maintains an instance variable that maintains the "modification count" for the collection. Every time any change is made to the collection, the modification count is incremented. When an iterator is created, the current value of the modification count is stored in the iterator. Any time an iterator accesses the collection, it first compares the value of the modification count it is holding with the value of the modification count in the collection. If the counts differ, an error is generated. Iterators that are designed to fail as soon as a problem is detected are referred to as **fast-fail iterators**.

The previous paragraph might have given the impression that you should not use iterators in your program. However, iterators are probably the most efficient means of stepping through the elements in a collection. The problems we described only occur when you try to remove elements from a collection that has an active iterator. If you need to remove elements from a collection, either make sure there are no active iterators or use the `remove()` method in the iterator class to remove the elements instead of the collection's `remove()` method.

We have spent a good deal of time describing the `Collection` interface, but as mentioned earlier, the Java Collection Framework does not provide any direct implementation of this interface because it is too general for most applications. We need to create subinterfaces that have characteristics of the data structures described in Chapters 6 through 8. The first one we discuss is the `Set` interface of Figure 9-1.

9.3.2 The `Set` Interface

As you learned in Chapter 8, a **set** is a collection of unique elements that is position independent. To implement this idea in the Java Collection Framework, the `Set` interface extends `Collection` but explicitly forbids duplicates. Two sets are considered equal if they contain exactly the same elements, regardless of where they appear within the collection.

The `Set` interface contains only the methods inherited from `Collection`. Therefore, it looks exactly like the interface in Figure 9-2, but with more specific semantics regarding the allowable behavior of the interface methods. These set-related behaviors are described in Figure 9-5.

METHOD	METHOD BEHAVIOR IN THE `Set` INTERFACE
`boolean add(E element)`	Adds `element` to the collection only if it is not currently there
`boolean remove(Object target)`	Removes `target` from the set
`boolean contains(Object target)`	Returns true if `target` appears anywhere in the set
`boolean containsAll(Collection<?> c)`	The subset operation defined in Section 8.1.1
`boolean addAll(Collection<? extends E> c)`	The set union operation in Figure 8-2
`boolean retainAll(Collection<?> c)`	The set intersection operation in Figure 8-2
`boolean removeAll(Collection<?> c)`	The set difference operation in Figure 8-2
`Object[] toArray()`	Places the elements of the set into an array in arbitrary order

[FIGURE 9-5] Behavior of the methods in the `Set` interface

To illustrate how to use a set, consider the problem of trying to remove duplicate items from a collection. One way would be to remove the elements from the collection and put them into a set. Because a set does not allow duplicates, the duplicate elements in the collection would not be added. The set would then contain all of the unique items from the collection with the duplicates automatically removed. Now we would simply copy the items in the set back into the original collection.

The method `removeDuplicates()` in Figure 9-6 uses this technique to remove duplicates from an arbitrary collection. (The method uses an implementation of sets called a `HashSet`, which we discuss at length in Section 9.4.1.1.)

```
public <T> void removeDuplicates( Collection<T> aCollection ) {
    // The set that will remove the duplicates
    Set<T> noDups = new HashSet<T>();

    // Step through the elements in the collection using an
    // iterator. Add each element to the set. If the element is
    // already in the set, it will not be added again because
    // sets do not allow duplicates.
    for ( T element : aCollection ) {
        noDups.add( element );
    }

    // Remove all elements from the collection
    aCollection.clear();

    // Now step through the elements in the set and add them
    // to the empty collection. The result will be the original
    // collection without duplicates.
    for ( T element : noDups ) {
        aCollection.add( element );
    }
}
```

[FIGURE 9-6] Removing duplicates from a collection

Fortunately, the collection classes also provide bulk operators, which eliminate the need to place elements from the collection into the set, one at a time, and then step through the set again, putting elements back into the collection. Using these bulk methods, you can rewrite the `removeDuplicates()` method using only three statements (see Figure 9-7).

```
public <T> void removeDuplicates( Collection<T> aCollection ) {
    // Create a set that contains all of the unique elements
    // from the collection
    Set<T> noDups = new HashSet<T>( aCollection );

    // Remove all elements from the collection
    aCollection.clear();
```

continued

```
    // Put all of the elements in the set back into the collection
   aCollection.addAll( noDups );
}
```

[FIGURE 9-7] Removing duplicates using the bulk operations

As a second example of the usefulness of sets, let's consider the method `numUnique()` in Figure 9-8, which returns the number of unique elements in an arbitrary collection called `aCollection`. The method creates a new set from the collection and then uses the `size()` method to return the number of elements contained in the newly constructed set. You can accomplish all of this in a single line of code, which is a good demonstration of the power of the Java Collection Framework.

```
public <T> int numUnique( Collection<T> aCollection ) {
    return new HashSet<T>( aCollection ).size();
}
```

[FIGURE 9-8] Determining the number of unique items in a collection

To summarize, a set is an unordered collection of elements that does not contain duplicate values. A set is said to be unordered because the order of the elements does not matter. In other words, the sets {1,2,3}, {2,3,1}, and {3,1,2} are equivalent. This makes it impossible to identify an element in a set based on its location—in other words, you can ask if 1 is in a set, but you cannot ask if 1 is the first element in a set.

As you saw in Chapter 6, a list is a data structure that differs from a set in two fundamental ways: It is an ordered collection, and it can contain duplicates. The next section discusses the `List` interface.

9.3.3 The `List` Interface

The `List` interface in the Java Collection Framework inherits all of the methods in the `Collection` interface of Figure 9-2, but adds more methods to support the two special behaviors of lists. Specifically, these new methods allow the following operations:

- *Positional access*—As we described in Section 6.2.2, we can access an element in a list using either a current position indicator (now termed an *iterator*) or by its position number within the list. The `List` interface includes methods for implementing this latter type of positional access.

- *Positional search*—We can search for a specific object in a list and return the position number in the list where that object was found.
- *Extended iteration*—The `List` interface includes iteration methods that explicitly take advantage of the inherent linear ordering of lists.
- *Subrange operations*—These methods allow us to perform operations on any subrange of the list—that is, on elements in the list in positions [*m*...*n*], for any legal values of *m*, *n*.

The `List` interface of the Java Collection Framework is shown in Figure 9-9.

```
public interface List<E> extends Collection<E> {
    // Positional access
    E get(int index);
    E set(int index, E element);                           // Optional
    void add(int index, E element);                        // Optional
    E remove(int index);                                   // Optional
    boolean addAll(int index, Collection<? Extends E> c); // Optional

    // Positional search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Extended iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // List subranges
    List<E> subList(int from, int to);
}
```

[FIGURE 9-9] `List` interface

The `get()`, `set()`, `add()`, and `remove()` methods in Figure 9-9 work almost exactly like their counterparts described in Chapter 6 and included in the `List` interface of Figure 6-6.

The methods in Figure 9-9 are in addition to the 13 methods that `List` *inherits from the* `Collection` *interface. For example, the* `add(o)` *method inherited from* `Collection` *adds its element to the end of the list, and the* `remove(o)` *method inherited from* `Collection` *deletes the first occurrence of its element from the list.*

The program in Figure 9-9a illustrates how to use many of the methods specified by the `List` interface. This program creates a list, fills it with numbers, and then performs some basic operations on the list.

```
import java.util.*;

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

        // Fill the list with the numbers 0 through 24
        for ( int i = 0; i < 25; i++ ) {
            nums.add( i );
        }

        // Remove the even numbers from the list
        for ( int i = 0; i < nums.size(); i++ ) {
            if ( nums.get( i ) % 2 == 0 ) {
                nums.remove( i );

                // Remove will shift all of the elements to the
                // right of the element being removed; the
                // loop control variable must be decremented so
                // the next element in the list is not skipped
                i--;
            }
        }

        // Multiply each number in the list by 2
        for ( int i = 0; i < nums.size(); i++ ) {
            nums.set( i , nums.get( i ) * 2);
        }

        // Print the resulting list
        System.out.println( nums );
    }

}  // ListExamples
```

[FIGURE 9-9a] Using a `List`

Most of the code in Figure 9-9a is straightforward. The only surprise might be the loop that removes the even numbers from the list. When you invoke `remove(i)`, not only is the element at position `i` removed from the list, any elements at positions greater than `i` are shifted down one position. The element that was at position `i` + 1 is moved to position `i`, the element at position `i` + 2 is moved to position `i` + 1, and so on. We could avoid the need to decrement `i` every time we remove an element from the list by working from the end of the list to the beginning, as shown in the following code:

```
for ( int i = nums.size() - 1; i >= 0; i-- ) {
    if ( nums.get( i ) % 2 == 0 ) {
        nums.remove( i );
    }
}
```

The `addAll()` method of Figure 9-9 adds all the elements of a collection to the list beginning at the specified position. The `indexOf()` and `lastIndexOf()` methods locate an occurrence of an object within the list. The `indexOf()` method returns the index of its first occurrence in the list; `lastIndexOf()` returns the index of the last occurrence.

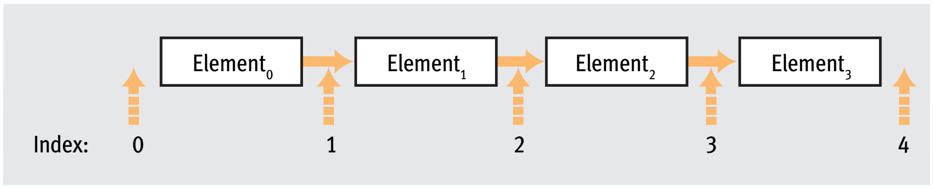
Lists provide an additional type of iterator object called `ListIterator`, which is a subclass of the `Iterator` interface in Figure 9-4. `ListIterator` provides a much richer set of positioning and movement operations that are appropriate for list data structures. The specifications for the `ListIterator` interface are shown in Figure 9-10.

```
public interface ListIterator<E> extends Iterator<E> {
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void remove();          // Optional
    void set(E o);          // Optional
    void add(E o);          // Optional
}
```

[FIGURE 9-10] `ListIterator` interface

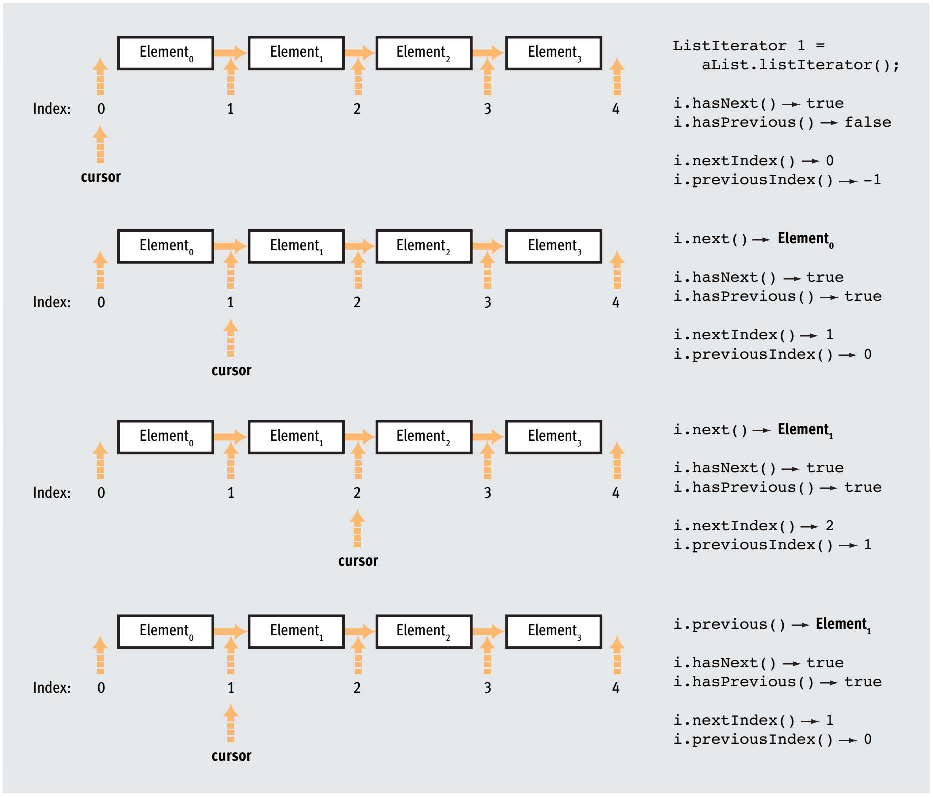
The `ListIterator` methods defined in the `List` class of Figure 9-9 allow you to obtain a `ListIterator` object that is either positioned at the first element in the list (no parameters) or at the element whose position number is specified by the integer parameter. A `ListIterator` can iterate forward and backward through a list, and allows you to obtain the iterator's current position in the list. From a user's point of view, a `ListIterator` behaves like the doubly linked list structure that we presented in Section 6.2.3.3. The `next()` and `hasNext()` methods allow you to traverse a list from the first element to the last, whereas the `previous()` and `hasPrevious()` methods let you traverse from last to first.

When dealing with a `ListIterator`, it can be useful to think of the cursor as not directly referring to one of the nodes in the list but as lying between two nodes in the list (see Figure 9-11).



[FIGURE 9-11] Cursor locations in a `ListIterator`

Invoking `next()` on a `ListIterator` object returns the element to the right of the current cursor and advances the cursor forward one position. Invoking `previous()` returns the element to the left of the cursor and moves the cursor back one position. Note that invoking `next()` immediately after an invocation of `previous()` will return the same element. The `nextIndex()` method returns the index of the list element that will be returned by the next invocation of `next()`, and correspondingly, the `previousIndex()` method returns the index of the list element that will be returned by the next invocation of `previous()`. Figure 9-12 illustrates example invocations of these iterator methods on a list and the values they return.



[FIGURE 9-12] Using the `ListIterator` methods

The `ListIterator` interface includes methods that modify the list over which it is iterating. The `remove()` method for a `ListIterator` removes the element just returned by either `next()` or `previous()`, similar to the behavior of the `remove()`method in the `Iterator` interface. For example, we could modify the loop that removes the even numbers from the list in Figure 9-9a to use a `ListIterator`, as shown in the following code:

```
ListIterator<Integer> iter = nums.listIterator();
while ( iter.hasNext() ) {
    if ( iter.next() % 2 == 0 ) {
        iter.remove();
    }
}
```

The add() method in the ListIterator class behaves in exactly the same way as the add() method in the Iterator class. It inserts the new element into the list before the element that would be returned by next() and after the element that would be returned by previous(). If the list is empty, the new element becomes the only element in the list. The element is inserted before the cursor, which means that next() is not affected, while a call to previous() returns the element just added to the list. The set() method is used to replace the last element returned by next() or previous() in the list. For example, the code in Figure 9-9a that multiplies each element in the list by 2 could be rewritten to use a ListIterator, as shown in the following code:

```
ListIterator<Integer> iter = nums.listIterator();
while ( iter.hasNext() ) {
    iter.set( iter.next() * 2 );
}
```

To further illustrate the use of a ListIterator, consider the problem of removing duplicate elements from a list. The technique in Figure 9-7 to remove duplicates from an arbitrary collection does not work for a list because we cannot guarantee anything about the order in which the elements are copied from the set back into the list. However, the program must preserve the ordering of the elements in the list if it is to work correctly.

The method in Figure 9-13 removes duplicates from a list using a set and the remove() method in the ListIterator class. The program uses a ListIterator to step through the elements in the list. As the iterator examines each element, the element is added to the set. If the element cannot be added to the set, then it has been seen already, and it is removed from the list.

```
public static <T> void removeDuplicates(List<T> aList ) {
    Set<T> unique = new HashSet<T>();
    ListIterator<T> i = aList.listIterator();

    while( i.hasNext() ) {
      if ( !unique.add( i.next() ) ) {
          i.remove();
      }
    }
}
```

[FIGURE 9-13] Removing duplicates from a list

The last `List` method we examine in this section is `subList()`. The `List` returned by the method `subList(int m, int n)` includes all elements in the original list from position `m` inclusive to position `n` exclusive. For example:

```
List<Strings> pets = new ArrayList<String>();
pets.add( "Grumpy" );
pets.add( "Mouse" );
pets.add( "Reiker" );
pets.add( "Roxy" );
pets.add( "Buster" );

// List cats will contain: Grumpy, Mouse, Reiker
List<String> cats = pets.sublist( 1,4 );

// List dogs will contain: Roxy, Buster
List<String> dogs = pets.sublist( 4,6 );
```

The value returned does not represent a new list object but simply a new "view" of the original list that includes the specified elements. Therefore, any changes to the list returned by the sublist method are effectively changes to the original list itself. For example, executing the statement `dogs.clear()` removes the strings `"Roxy"` and `"Buster"` from the list `pets`.

The `IntPriorityQueue` class of Figure 9-14 illustrates how you can use a list to implement the priority queue data structure introduced in Section 6.4.4. The list stores the integers in the queue in priority order—the integer with the highest value is stored in position 0 of the list, and the item with the lowest value is stored at the end. The `enqueue()` method uses a `ListIterator` to determine where to place the new item in the list.

```
import java.util.*;

/**
 * A priority queue that holds int values—implemented using a list.
 */
public class IntPriorityQueue {
    private List<Integer> queue; // The actual queue

    /**
     * Create a new priority queue.
     */
```

continued

```
    public IntPriorityQueue() {
      queue = new LinkedList<Integer>();
    }

    /**
     * Add an element to the queue.
     *
     * @param val the element to add to the queue.
     */
    public void enqueue( int val ) {
      ListIterator<Integer> i = queue.listIterator();
      int loc = 0;

      // Find the first item with a lower value than the new one
      while( i.hasNext() && val < i.next() ) {
          loc = i.nextIndex(); // The last item we examined in the list
      }

      // loc is the location of the first item with a lower value
      queue.add( loc, val ); }

    /**
     * Return an item with the highest priority.
     *
     * @return an element with the highest priority.
     */
    public int dequeue() {
      return queue.remove( 0 ); // Highest value is always at the front
    }

    /**
     * Determine if the queue is empty.
     *
     * @return true if the queue is empty and false otherwise.
     */
    public boolean isEmpty() {
      return queue.isEmpty();
    }
} // IntPriorityQueue
```

[FIGURE 9-14] A priority queue based on a `List`

To summarize the characteristics of the interfaces we have introduced so far, we can say that the `Collection` interface represents an arbitrary group of objects. The `Set` interface extends Collection by forbidding duplicate elements in the collection. Finally, the List interface extends Collection by allowing both duplicates and positional indexing and accessing of elements.

9.3.4 The Map Interface

In Section 8.2, we introduced the **map** data structure, also called a **key-access table**. We showed that a map is a special type of set in which the members of the collection are not individual elements but 2-tuples of the form (*key*, *value*), where every key is unique. This pair represents a mapping from key objects to value objects in the sense that, given a unique key, we can use the key to locate its associated value object. Some examples of mappings might be:

- A map of student ID numbers to student database records
- A dictionary in which words are mapped to their meanings
- A mapping from values in base 2 to their value in base 10

The primary difference between a Collection and a Map is that in a Collection you add, remove, and look up individual items, whereas in a Map you add, remove, and look up items using a key-value pair. The typical use of a map data structure is to provide direct access to values stored by key.

Because maps access elements quite differently from the techniques used in Collection, the Map interface does not extend the Collection interface but stands on its own. The Map interface of the Java Collection Framework is listed in Figure 9-15.

```
public interface Map<K,V> {
    // Basic map operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk operations
    void putAll(Map<? extends K, ? extends V> t);
    void clear();
```

continued

```
    // Collection views
    Set<K> keySet();
    Collection<V> values();
    Set<Map.Entry<K,V>> entrySet();

    // Interface for entrySet elements
    interface Entry<K,V> {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

[FIGURE 9-15] `Map` interface

The behavior of the five map operations `put()`, `get()`, `remove()`, `isEmpty()`, and `size()` shown in Figure 9-15 is virtually identical to that of the methods described in Section 8.2 and included in the `Table` interface of Figure 8-5. The `put()` method adds a (*key*, *value*) tuple to the map, `remove()` deletes the tuple with the given key field if it is present in the map, and `get()` returns the value field associated with a given key field if such a key field exists. The primary difference between the `Table` interface of Chapter 8 and the `Map` interface of Figure 9-15 is that the `put()` and `remove()` methods in the `Map` interface return a reference to the value that was previously associated with this key.

If you do not allow **null** values to be placed in a map, you can use the return values from the `put()` and `remove()` methods to indicate whether an element was in the map. For `put()`, a return value of **null** indicates that the key was not in the map and had no previous value field associated with it. A non-null value indicates that a new value has been associated with an existing key, and the value that was returned was the previous value field. A return value of **null** from `remove()` indicates that the key was not in the table (in other words, no value is associated with the key). Many programmers use their methods in this way. A safer way to determine if a key or a value is in a map is to use the `containsKey()` or `containsValue()` methods. The `containsKey()`method is efficient because there is a direct mapping between a key and its location within a map. (This mapping is specified by the hashing function.) However, the `containsValue()` method is significantly slower because there is no quick way to locate a value field, except from its mapping via the key. Therefore, this method must iterate through the entire map looking for this value, an O(n) linear time operation, where n is the number of tuples in the map.

The Map interface in Figure 9-15 is also much richer than the interface shown in Figure 8-5, and includes additional helpful methods. The putAll() method is the equivalent of the addAll() method in the Set interface. It adds all of the mappings in one map into another, ensuring that all key values remain unique. The clear() method does the obvious—it removes all the (*key*, *value*) tuples from the map.

An interesting group of methods in Figure 9-15 are the three "collection views." Because a map is a set of tuples rather than individual elements, it does not make much sense to talk about "iterating" through a map. Would we iterate through the keys? The values? Both? To give meaning to the concept of iterating through a map, the interface allows you to restructure the map to view it as different types of collections. For example, the keySet() method allows you to view your map as a set that contains all the keys present in the original map structure. Thus, if our map contained the four tuples (1, 2), (8, 14), (5, 2), and (17, 33), then the set returned from a call to keySet() would be s = {1, 8, 5, 17}, although the exact ordering is immaterial. We could now invoke the iterator() method on s and iterate through the keys in the map.

The method values() in Figure 9-15 returns a collection that includes all the values contained in the map. Using the same four tuples given earlier, the Collection object returned by a call to values() would be (2, 14, 2, 33). Note that this object is not a set, because different keys (in our case, 1 and 5) can map to the same value—in our case, 2. Finally, the method entrySet() returns a set whose elements are the (*key*, *value*) pairs contained in the map. The Map interface includes an inner interface, called Entry, that represents the data type of the elements in the set returned by the entrySet() method.

The BookShelf class of Figure 9-16 illustrates the use of a Map. The BookShelf class allows a user to store book titles by their unique ISBN (International Standard Book Number). A Map, called theShelf, holds the ISBNs (keys) and the titles (values). Most of the methods in the BookShelf class are simple wrappers around the corresponding Map methods. The toString() method uses an iterator to iterate through the keys stored in the map. These keys then retrieve their associated titles.

```
import java.util.*;

/**
 * A class that represents a bookshelf. Books on the shelf
 * are stored and retrieved by ISBN.
 */
public class BookShelf {
    private Map<String, String> theShelf;  // The books
```

continued

```
/**
 * Create a new book shelf.
 */
public BookShelf() {
    theShelf = new HashMap<String, String>();
}

/**
 * Add a book to the bookshelf. The book is added only if a book
 * with the same ISBN is not already on the shelf.  The return
 * value indicates whether the book was added to the shelf.
 *
 * @param isbn the ISBN of the book to be added.
 * @param title the title of the book.
 * @return true if the book was added to the shelf.
 */
public boolean addTitle( String isbn, String title ) {
    // If the ISBN is a key in the map, the book is on the shelf
    boolean onShelf = theShelf.containsKey( isbn );

    // If the book is not already there, add it
    if ( !onShelf ) {
        theShelf.put( isbn, title );
    }

    return onShelf;
}

/**
 * Return the title associated with the given ISBN.
 *
 * @param isbn the ISBN of the title to be retrieved.
 * @return the title or null if the ISBN is not on the shelf.
 */
public String getTitle( String isbn ) {
    return theShelf.get( isbn );
}

/**
 * Remove the specified title from the shelf. The return value
 * indicates whether the title was removed from the shelf.
 *
```

continued

```
     * @param isbn the ISBN of the title to remove.
     * @return true if the title was removed from the shelf.
     */
    public boolean removeTitle( String isbn ) {
        // Remove returns null if the key is not in the map
        return theShelf.remove( isbn ) != null;
    }

    /**
     * Return a list that contains all of the titles on the shelf.
     *
     * @return a list that contains all of the titles on the shelf.
     */
    public List getAllTitles() {
        // Values returns a collection—convert it to a list
        return new ArrayList<String>( theShelf.values() );
    }

    /**
     * Return a string that contains all the ISBNs and the titles.
     *
     * @return a string representation of the books on the shelf.
     */
    public String toString() {
        // Use a string buffer for efficiency
        StringBuffer books = new StringBuffer( "The books:\n" );

        for ( String key : theShelf.keySet() ){
            String isbn = key;

            // Use the ISBN to get the corresponding title
            books.append( "   " + isbn + "   " +
                          theShelf.get( isbn ) + "\n" );
        }

        return books.toString();
    }
}
```

[FIGURE 9-16] `BookShelf` class

9.3.5 Sorted Interfaces

9.3.5.1 The Natural Ordering of Objects

In Figure 9-1, we saw that the Java Collection Framework includes two sorted interfaces, `SortedSet` and `SortedMap`, that allow you to retrieve objects stored in their collections in sorted order. However, before we can discuss the behavior of these interfaces, we need to know how a collection sorts its object. Clearly, for objects such as numbers or strings, the answer is easy—you sort them numerically or alphabetically. However, what about a collection of objects that represents something like first and last names or breeds of dogs? Now the natural order of these objects is not so obvious.

Just as the `equals()` method in class `Object` provides a way to determine if two objects are equal, we need a method that defines the relative ordering of two objects. The Java API defines the interface `Comparable` for this purpose. The interface specifies only a single method, `compareTo()`, which you can use to determine if an object is less than, equal to, or greater than another object. The `Comparable` interface is shown in Figure 9-17.

```
public interface Comparable<T> {
    int compareTo( T o );
}
```

[FIGURE 9-17] `Comparable` interface

The `compareTo()` method returns an integer value that indicates the relative ordering of the objects being compared. For example, the method invocation `aString.compareTo( anotherString );` returns a negative value if `aString` is strictly less than `anotherString`, zero if `aString` is equal to `anotherString`, and a positive number if `aString` is strictly greater than `anotherString`. Note that the specification of `compareTo()` does not specify the magnitude of the return value, only its sign. Thus, the following `if` statement:

```
if ( aString.compareTo( anotherString ) == -1 ) {
    // Code to execute if aString < anotherString
}
```

might not correctly detect all of the situations when `aString` is less than `anotherString`. The correct way to make the comparison is to consider only the sign of the return value and not its magnitude:

```
if ( aString.compareTo( anotherString ) < 0 ) {
    // Code to execute if aString < anotherString
}
```

The `compareTo()` method is referred to as a class's **natural comparison method**, and the ordering it imposes on the objects of the class is referred to as the **natural ordering** of the class.

Writing a `compareTo()` method for a class is easy. Consider the `Name` class of Figure 9-18, which consists of a first and last name. The class defines the methods `equals()` and `compareTo()`. Two names are considered equal if their first and last names are the same. The `compareTo()` method in Figure 9-18 defines a natural ordering in which names are ordered by last name; in cases where the last names are the same, the ordering is by first name. Note that the `Name` class implements the `Comparable` interface.

```
/**
 * A class that represents a name consisting of a
 * first and last name
 */
public class Name implements Comparable<Name>{
    String first, last;  // The name

    /**
     * Create a new name.
     *
     * @param firstName the first name.
     * @param lastName the last name.
     */
    public Name( String firstName, String lastName ) {
        first = firstName;
        last = lastName;
    }

    /**
     * Get the first name.
     *
```

continued

```
     * @return the first name.
     */
    public String getFirst() {
        return first;
    }

    /**
     * Get the last name.
     *
     * @return the last name.
     */
    public String getLast() {
        return last;
    }

    /**
     * Compare two names and determine their relative order.
     *
     * @param otherName the name to compare this name to.
     * @return a negative number if this name is less than the given
     *         name, zero if the names are equal, and a positive number
     *         if this name is greater than the specified string.
     */
    public int compareTo( Name otherName ) {
        // Compare the last names using the compareTo() method
        // defined in the string class.
        int retVal = last.compareTo( otherName.getLast() );

        // If the last names are equal, compare the first names.
        if ( retVal == 0 ) {
            retVal = first.compareTo( otherName.getFirst() );
        }

        return retVal;
    }

    /**
     * Determine if this name is equal to the specified object.
     *
     * @param otherObject the object to compare this name to.
     * @return true if this name is equal to the specified object.
     */
```

continued

```
    public boolean equals( Object otherObject ) {
        boolean retVal = false;

        // If this method is passed a reference to something
        // other than a name, the objects are not equal.
        if ( otherObject instanceof Name ) {
            Name otherName = (Name)otherObject;

            // Equal if their first and last names are the same.
            retVal = last.equals( otherName.getLast() ) &&
                     first.equals( otherName.getFirst() );
        }

        return retVal;
    }
}
```

[FIGURE 9-18] Defining a `compareTo()` method

You must follow several rules when writing a `compareTo()` method. First, the sign of the result returned by an invocation of `x.compareTo(y)` must be the opposite of the sign returned by an invocation of `y.compareTo(x)`. Second, the `compareTo()` method must be transitive. In other words, if `x.compareTo(y)` $>$ 0 and `y.compareTo(z)` $>$ 0, then `x.compareTo(z)` $>$ 0. Finally, if `x.equals(y)` is true, then the results returned by the invocation of `x.compareTo(y)` and `y.compareTo(x)` must both be 0.

The reason for the last rule is that a class that implements the `Comparable` interface uses two methods to determine if objects are equal. You could invoke either `equals()` or `compareTo()`. You must write the `equals()` method and the `compareTo()` method to be consistent. In other words, if `equals()` returns true for two objects, then `compareTo()` should return 0 for the same two objects. Writing a class in which the `compareTo()` method is inconsistent with the `equals()` method can cause unpredictable behavior, because you can never be certain when someone uses your class whether they will use the `equals()` or the `compareTo()` method to determine if two objects from the class are the same. Both methods are perfectly valid.

Some classes have no natural ordering or may have multiple orderings. For example, consider the `Dog` class of Figure 9-19. The state of a dog consists of its name, breed, and gender. The `equals()` method for this class does what you would expect: it compares the name, breed, and gender of the dogs and returns true if all three values are identical.

```java
/**
 * A class that represents a dog. The state of a dog consists of its
 * name, breed, and gender. Note that this class is not comparable.
 */
public class Dog {
    private String breed, name, gender;  // State for the dog

    /**
     * Create a new dog object.
     *
     * @param theBreed the breed of the dog
     * @param theName the name of this dog
     * @param theGender the gender of this dog
     */
    public Dog( String theBreed, String theName, String theGender ) {
        breed = theBreed;
        name = theName;
        gender = theGender;
    }

    /**
     * Return the breed of this dog
     *
     * @return the breed of this dog
     */
    public String getBreed() {
        return breed;
    }

    /**
     * Return the name of this dog
     *
     * @return the name of this dog
     */
    public String getName() {
        return name;
    }

    /**
     * Return the gender of this dog
     *
     * @return the gender of this dog
     */
```

continued

```
    public String getGender() {
        return gender;
    }

    /**
     * Indicates whether some other object is equal to this one.
     *
     * @param o the object to be compared with
     * @return true if this object is the same as the argument.
     */
    public boolean equals( Object o ) {
        boolean retVal = false;

        if ( o instanceof Dog ) {
            Dog other = (Dog)o;
            retVal = breed.equals( other.getBreed() ) &&
                     name.equals( other.getName() ) &&
                     gender.equals( other.getGender() );
        }

        return retVal;
    }
}
```

[FIGURE 9-19] `Dog` class

However, we have many options for comparing two `Dog` objects to see which one is greater. We simply could compare the dogs by their name, or we might want to use some combination of their name and breed. The problem with these approaches is the difficulty of writing a `compareTo()` method that is consistent with the `equals()` method. In such situations, it is probably best to design a class that does not implement the `Comparable` interface.

A **comparator** can provide a natural ordering for classes that do not implement the `Comparable` interface or for classes in which you can order objects in more than one way. A `Comparator` is an object that encapsulates an ordering, much like an external iterator is an object that encapsulates an iteration over a collection. The `Comparator` interface, like the `Comparable` interface, defines a single `compare()` method, except that the `compare()` method of the `Comparator` interface requires two object references when it is invoked instead of one. The `Comparator` interface is shown in Figure 9-20.

```
public interface Comparator<T> {
    int compare( T o1, T o2 );
}
```

[FIGURE 9-20] Comparator interface

Figure 9-21 shows two comparators you could use with the Dog class. The first comparator provides an ordering based on the dog's name. The second comparator implements an ordering based first on the breed and then by the dog's name.

```
public class DogByName implements Comparator<Dog> {
    public int compare( Dog d1, Dog d2 ) {
        return d1.getName().compareTo( d2.getName() );
    }
}

public class DogByBreedAndName implements Comparator<Dog> {
    public int compare( Dog d1, Dog d2 ) {
        int retVal = d1.getBreed().compareTo( d2.getBreed() );
        if ( retVal == 0 ) {
            retVal = d1.getName().compareTo( d2.getName() );
        }

        return retVal;
    }
}
```

[FIGURE 9-21] Comparators for the Dog class

Both comparators in Figure 9-21 have the same look and feel as the compareTo() method of the Comparable interface. The major difference is that two references are required to identify the objects being compared instead of one. The next section discusses how to use Comparable objects and Comparator objects with the sorted collections of the Java Collection Framework.

9.3.5.2 Sorted Sets and Sorted Maps

A SortedSet is a set that maintains its elements in ascending order according to either the natural ordering of its elements or a comparator provided when the set is created. The methods provided by a SortedSet behave exactly as the methods provided by a set, except that the SortedSet guarantees two things: its iterator traverses the elements of the set in ascending order, and the array returned by the toArray() method contains the set's elements stored in ascending order. The elements in a SortedSet must implement the Comparable interface, or a Comparator must be provided to the set at creation time to specify how to order the elements.

The SortedSet interface provides additional operators that take advantage of the ordering of its elements, as shown in Figure 9-22.

```
public interface SortedSet<E> extends Set<E> {
    // Range view
    SortedSet<E> subSet(E from, E to);
    SortedSet<E> headSet(E to);
    SortedSet<E> tailSet(E from);

    // Endpoints
    E first();
    E last();

    // Comparator
    Comparator<? super E> comparator();
}
```

[FIGURE 9-22] SortedSet interface

The three range-view operations provided by the SortedSet interface return a subset of the elements from the original set that fall within the specified boundaries. The subSet() method takes two endpoints and returns a subset of all the elements that are greater than or equal to the from endpoint and are strictly less than the to endpoint. The methods headSet() and tailSet() work in a similar fashion, but one of their endpoints is the first or last element, respectively. A range view of a sorted set is a window into whatever portion of the set lies in the designated range. Changes made to the range view change the original SortedSet, and vice versa. The first() and last() methods of the SortedSet interface return the first (smallest) and last (largest) elements in the set, respectively.

To see what you can do using the range-view operators, imagine a set that contains the serial numbers of the computing equipment at a university. The serial numbers consist of five characters. The first character is a letter that identifies the vendor, and the remaining four characters are digits that uniquely identify a specific piece of equipment from that vendor. Assume that the variable serialNumbers refers to a SortedSet that contains all of the serial numbers at the university. You could determine the number of machines manufactured by the vendor identified by the character *A* using the following line of code:

```
int num = serialNumbers.subset( "A0000", "B0000" ).size();
```

The subset returned by the `subset()` method is a `SortedSet` containing all serial numbers that start at `"A0000"` and run up to, but do not include, serial number `"B0000"`. The size of this subset is the number of machines manufactured by vendor A.

As a second example, you could remove all of vendor A's equipment from the set by executing the following line of code:

```
serialNumbers.subset( "A0000", "B0000" ).clear();
```

Again, the `subset()` method returns all of the serial numbers that correspond to vendor A in a `SortedSet`. The `clear()` method removes these elements from the subset, but because the subset is simply a window onto the original `SortedSet`, the elements are removed from the sorted set as well.

A `SortedMap` is similar to a `SortedSet`. A `SortedMap` is a map whose keys are stored in ascending order based on the natural ordering of the keys. Like the `SortedSet`, you can use a comparator to determine order if the keys do not implement the `Comparable` interface or if you want to use an ordering besides the natural order of the keys. The `SortedMap` interface is shown in Figure 9-23; its methods behave like those in the `SortedSet`.

```
public interface SortedMap<K,V> extends Map<K,V> {
    // Range view
    SortedMap<K,V> subMap(K from, K to);
    SortedMap<K,V> headMap(K to);
    SortedMap<K,V> tailMap(K from);

    // Endpoints
    K firstKey();
    K lastKey();

    // Comparator
    Comparator<? super K> comparator();
}
```

[FIGURE 9-23] `SortedMap` interface

THE FIRST COMPUTER BUG

In 1945, Grace Murray Hopper was part of a team of computer scientists working on the Harvard University Mark II computer. Unlike modern computers, the Mark II used relays, or electrical switches, as its basic components. The team had been having problems with the computer and were trying to determine the cause, but the task proved difficult because the problems were intermittent.

On September 9, when the machine was having problems again, Hopper looked inside the machine and discovered that a dead moth had been trapped between the contacts of a relay. She removed the moth and the problem disappeared. She taped the moth to a log book and made the following entry: "Relay #70, in Panel F (moth) in relay. First actual case of bug being found." People started to say that the computer had been *debugged*, and a new term was born. (See Hopper's biography near the end of this chapter.)

However, if you carefully read the wording of Hopper's log entry, it implies that the term *bug* already existed. Although Hopper may have found the first *computer* bug, the term had already been commonly used to describe a defect. The *Hawkin's New Catechism of Electricity*, published in 1896 by Theo, Audel & Company, contains the entry: "The term *bug* is used to a limited extent to designate any fault or trouble in the connections or working of electric apparatus."

9.4 Implementation Classes

The implementation classes reflect how the Java Collection Framework chooses to build classes that implement the interfaces presented in Section 9.3. That is, what types of structures—arrays, linked lists, trees, hash tables—are used to construct the sets, lists, and map collections contained in the framework?

Make sure you know the difference between an interface and its implementation. For example, Section 9.3.2 described a `List` interface that included such methods as `add()`, `remove()`, and `indexOf()`. We could implement that `List` interface using either an array, as we did in Section 6.2.3.1, or with a linked list, described in Section 6.2.3.2. This would represent a single interface with two distinct and separate implementations. When designing software, you should always think in terms of interfaces, not implementations. This way, your program is not dependent on idiosyncratic methods found in only one particular type of implementation, leaving you free to change implementations at a later time if desired.

When designing a piece of software, the most important issue is deciding what type of collection is best for holding your data elements. Once you decide, you can instantiate a variable of the proper interface type and place the elements into this object. For example, after deciding to use a linear data structure to hold the elements in your collection, you would do the following:

```
// Create an object that implements the List interface
// and place a reference to the object in the variable L
List<String> L = new ArrayList<String>() ;
```

You now design your program in terms of list objects and the standard list methods contained in the interface of Figure 9-9. In terms of correctness, it does not matter how you implement the list while developing your software. The only effect of the implementations you choose (although it can be an important one) is the performance you can obtain.

In this section, we focus on the so-called **general-purpose implementations** of the Java Collection Framework. These public classes represent the most general and flexible implementations that are used most of the time. The general-purpose implementations are summarized in Figure 9-24; the naming convention is <*implementation*><*interface*>, where <*implementation*> is the name of the data structure used in the implementation and <*interface*> is one of the standard interfaces in Figure 9-1. So, for example, the class name `HashMap` means that the class does a hash table implementation of the `Map` interface shown in Figure 9-15.

INTERFACE	IMPLEMENTATION				HISTORICAL
`Set`	`HashSet`		`TreeSet`		
`List`		`ArrayList`		`LinkedList`	`Vector`
					`Stack`
`Map`	`HashMap`		`TreeMap`		`HashTable`
					`Properties`

[FIGURE 9-24] The implementation classes

Note from Figure 9-24 that the Java Collection Framework provides two different implementations for each of the interfaces presented in Section 9.3, except for `Collection`, which has no implementations, as mentioned earlier. The two implementations of each interface typically perform differently on certain operations within the interface. One implementation may work well for operation `a()` but fare quite poorly with `b()`, while another implementation might behave in exactly the opposite manner. You must decide which implementation is most appropriate for a specific problem. Alternatively, you may decide to design and implement a totally new implementation. This decision requires a thorough understanding of the data structure concepts presented in Chapters 6, 7, and 8.

In addition to the default constructor, all collection implementations conventionally provide a one-parameter constructor that takes a `Collection` as its argument. The constructor initializes the object to contain all of the elements in the specified `Collection`. The sorted collections, `SortedSet` and `SortedMap`, provide two other standard constructors: one constructor takes a `Comparator` and one takes a sorted collection. The constructor that takes a `Comparator` creates a new sorted collection whose elements are sorted according to the specified comparator. The constructor that takes a sorted collection creates a new sorted collection that contain the elements in the given collection and are sorted by the same comparator.

9.4.1 Sets

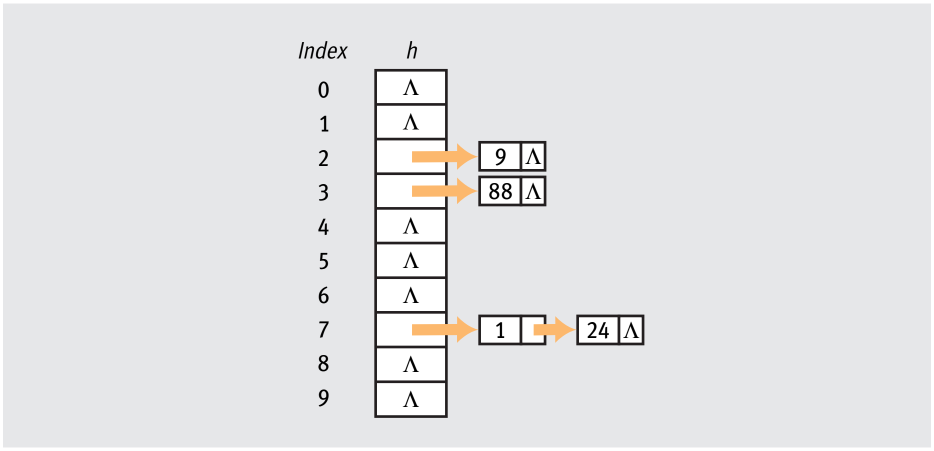
9.4.1.1 `HashSet`

Our first implementation of sets is `HashSet`, which uses the hash table techniques in Section 8.2.3 to implement a set data structure. In Chapter 8, we described how to use hashing to implement a map containing 2-tuples of the form (*key*, *value*). However, you can use the identical techniques described there to implement a `Set` simply by ignoring the value field. Because all the keys are unique, they form a set by definition.

One important decision by the designers of the Java Collection Framework was whether to use the open addressing (Figure 8-10) or chaining (Figures 8-12 and 8-13) method to handle collisions during hashing. They decided to use chaining, probably because it allows more than n values to be stored in a hash table of size n.

Using a `HashSet` implementation, the four-element set `s` = {24, 88, 1, 9} would be stored in a 10-element chained hash table `h`, as shown in Figure 9-25, assuming the following values were produced by our hash function f:

$$f(24) = 7, f(88) = 3, f(1) = 7, f(9) = 2.$$



[FIGURE 9-25] Implementation of a set using `HashSet`

As mentioned in Section 8.2.3, under best-case conditions, adding, retrieving, and removing elements from a `HashSet` are all O(1), the most efficient implementation possible. If the hash table, `h`, is quite sparse, very few elements in each of the linked lists will be referenced by `h[i]`—typically 0 or 1. To add a new element `e` to the set, we first search the linked list whose head pointer is stored in i = `f(e)` to see if the element is already there. If the table is very sparsely occupied, this takes only a single step. If it is not there, we add it to the head of the linked list, which takes a constant number of steps. Overall, then, the operation is O(1). The same analysis holds for deletions and retrievals. However, if E, the total number of elements stored in the hash table, is significantly larger than the size of the hash table n, typically referred to as the number of **buckets**, then the linked lists average E/n elements in length rather than 0 or 1, and the time to add, retrieve, and delete is $O(E/n)$.

Therefore, to make a `HashSet` implementation work reasonably well, you must *parameterize* it well. The two most important parameters for a successful implementation are (a) the **initial capacity** of the hash table, the value $n = 10$ in Figure 9-25; and (b) the **load factor** α. In Section 8.2.3, we defined the load factor as:

α = number of elements/size of the hash table

By this definition, the load factor for the table in Figure 9-25 would be:

$\alpha = 4/10 = 0.40$

However, the Java Collection Framework defines α in a slightly different way. In the framework, the load factor is defined as the *maximum* allowable ratio of the number of elements to the size of the hash table before we must resize and rebuild the hash table. That is, if $\alpha = 0.8$, it means that whenever α exceeds 0.8, we must enlarge the table (typically it is doubled) and then rehash all entries to find their proper location in the new table.

You can set both of these hash table parameters using the following two-parameter `HashSet` constructor:

```
HashSet(int initialCapacity, float loadFactor);
```

Other `HashSet` constructors allow you to use one or both of the default values for these numbers.

Higher values for `loadFactor` can save space and reduce the number of times that the hash table must be enlarged and rebuilt, a potentially time-consuming operation. However, you pay for this in terms of slower accesses and insertions, as the average length of the linked lists increases. Similarly, a larger `initialCapacity` helps to keep the load factor down, but requires additional space for the larger table. In addition, iterating through the elements of a set requires time proportional to $O(n + E)$, because we must not only visit every element of the set, we must check every entry in the hash table to determine whether it is empty. Thus, keeping the `initialCapacity` parameter small can improve the efficiency of iteration operations.

If you have a good estimate for *MAX*, the maximum number of elements you can place in the set, and this value is not too large, then a good rule of thumb is to make `initialCapacity` double the value of *MAX* and use the default value for `loadFactor`. Because you have about twice as many slots as set elements, the load factor will be about 0.5, which is well below its default value of 0.75. Therefore, no rebuilding or rehashing operations should ever be required. In addition, the length of each linked list is typically 0 or 1 (assuming that the hash function scatters elements reasonably well), so you should observe O(1) behavior for all accesses and insertions.

Notice the importance of knowing both algorithm analysis and the basic properties of hashing, such as collisions, load factors, and chaining. Even though the Java Collection Framework provides a `HashSet` implementation, we still must understand how to use it wisely by configuring it in the proper way. This is an example of why modern software developers need to understand data structures, as discussed in Chapters 6 through 8.

The last issue regarding the `HashSet` implementation is the hashing function. As you know, the performance of a hash table is highly dependent on a good hashing function that scatters keys evenly over the table. The root class of Java, `Object`, includes the following hashing function:

```
public int hashCode(); // Return a hash code for this object
```

Because every object in Java inherits from `Object`, every object has a `hashCode()` method that can be used by one of the hash implementations. The default `hashCode()` method defined in class `Object` returns distinct integers for distinct objects. This is usually accomplished by converting the internal address of the object into an integer.

The `hashCode()` and `equals()` methods have a relationship to which you must adhere for the hash implementations to work correctly. Specifically, if two objects are equal according to the `equals()` method, then calling the `hashCode()` method on each of the objects must produce the same value. If two objects are unequal according to the `equals()` method, the `hashCode()` methods for the two objects do not have to return distinct values.

If you think about how hashing works, you realize that these rules make perfect sense. Consider what might happen if two objects were equal according to the `equals()` method but had different hash code values. You would use the `hashCode()` method to search the bucket in which you would expect to find the object, but you would not find the object and incorrectly report that it was missing. On the other hand, the fact that `hashCode()` does not have to return distinct values for two unequal objects is precisely what causes a collision to occur in a hash table.

Essentially, these rules mean that whenever you override the `equals()` method, you must override `hashCode()` as well. The `Dog` class in Figure 9-26 illustrates how to override the `equals()` and `hashCode()` methods in a class.

```java
/**
 * A class that represents a dog. The state of a dog consists of its
 * name, breed, and gender. Note that this class is not comparable.
 */
public class Dog {

    // Code omitted

    /**
     * Indicates whether some other object is equal to this one.
     *
     * @param o the object to be compared with
     * @return true if this object is the same as the argument.
     */
    public boolean equals( Object o ) {
        boolean retVal = false;

        if ( o instanceof Dog ) {
            Dog other = (Dog)o;
            retVal = breed.equals( other.getBreed() ) &&
```

continued

```
                          name.equals( other.getName() ) &&
                          gender.equals( other.getGender() );
            }

            return retVal;
        }

        /**
         * Return a hash value for this object; the hash value for a dog
         * object is equal to the sum of the hash codes for the name,
         * breed, and gender of the dog.
         */
        public int hashCode() {
            return name.hashCode() + breed.hashCode() + gender.hashCode();
        }
    }
```

[FIGURE 9-26] Using the `Dog` class

9.4.1.2 TreeSet

The second implementation of sets in the Java Collection Framework is `TreeSet`, which uses a red-black tree to store the set elements. A red-black tree is a hierarchical data structure that we discussed in Section 7.5. You might recall that a red-black tree is a balanced binary search tree that does not permit a shape like the one in Figure 7-19a. It constantly maintains a balanced shape by performing a rotation, a restructuring of the tree around an individual element while maintaining the overall binary search tree property. This guarantees that inserting and retrieving elements can be completed in O(log2*n*) time.

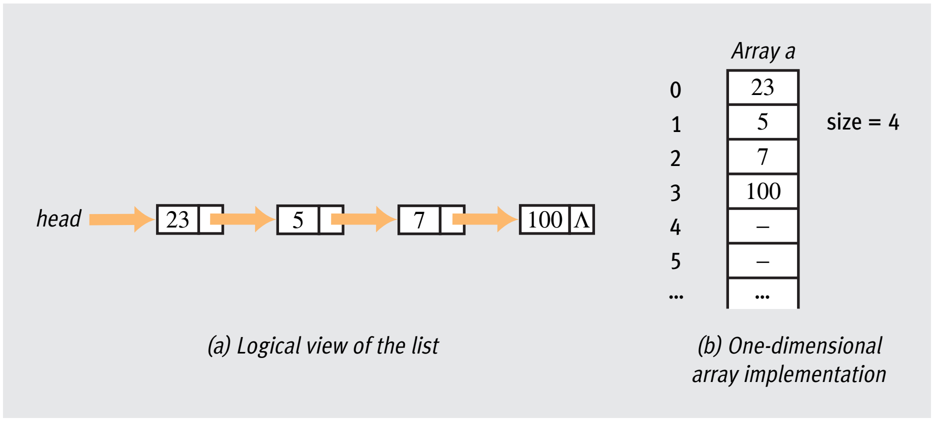
However, that doesn't seem very impressive. As we showed in the previous section, if you choose your parameters well, insertion and retrieval using a `HashSet` are both O(1). If speed is your most important consideration, you should definitely choose a `HashSet` implementation over the `TreeSet`, as most developers do.

So, what is the purpose of the `TreeSet` implementation? The answer lies in Figure 7-21, the tree sort algorithm. If *T* is a binary search tree, then an inorder traversal of *T* visits its elements in sorted order. Thus, if you must be able to iterate through the elements of a set in ascending or descending order, choose the `TreeSet`. Otherwise, use the `HashSet` for its superior speed and performance. Because it is extremely important to iterate in order through a `SortedSet`, the `TreeSet` implementation is the only one provided by the framework for this type of interface.

9.4.2 Lists

9.4.2.1 ArrayList

The `ArrayList` implementation of `List` uses a one-dimensional array structure to store the elements in the list. You implement this class like you implement an array-based list, as discussed in Section 6.2.3.1. The layout of the `ArrayList` implementation is shown in Figure 9-27.


head
23
5
7
100
Λ
Array a
0
1
2
3
4
5
...
100
–
size = 4
(a) Logical view of the list
(b) One-dimensional array implementation

[FIGURE 9-27] One-dimensional array implementation of a list

If we examine the structure shown in Figure 9-27b, we can quickly begin to appreciate its strengths and weaknesses. Positional access to an element in the list, the operation `get()`, is extremely fast due to the random-access nature of arrays. For example, retrieving the third element of the list using the linked structure of Figure 9-27a would require traversing the list beginning from `head`, an $O(n)$ operation. However, in the array implementation, we simply return the contents of `a[2]`, a constant-time $O(1)$ operation. Similarly, changing the information field of an individual node, the operation `set()`, is also $O(1)$. Adding a new element `e` to the end of the list is usually $O(1)$ as well because we can write the following:

```
a[size] = e;
size++;
```

Remember that duplicates are allowed in a `List`, so there is no need to first search the array to see if the element is there. However, the array might have reached maximum capacity before we attempt to add this new element; in that case, the array must be resized. When this occurs, the `ArrayList` implementation creates a new, larger array and copies all *n* elements from the old array into the new one. This requires O(*n*) time. Therefore, `add()` is O(1) if the array is not full and O(*n*) otherwise. (Although linear in time, the resizing operation is actually quite fast because of the method `System.arraycopy()`, which efficiently copies an entire array.) If accessing or modifying elements by position and adding new elements to the end of the list are common operations, then an `ArrayList` implementation works well, and would be the implementation of choice.

When we create the `ArrayList` representation, the following constructor allows us to specify the initial capacity of the array:

```
ArrayList(int initialCapacity); // initial array size
```

If you have a good estimate for the maximum number of elements in the list, create your array with an `initialCapacity` that is slightly larger than this maximum. Then you should never need a resizing and recopying operation.

The program in Figure 9-28 can measure the performance of random accesses within an array and linked list. The program creates two lists and fills them with random numbers. It then measures the amount of time required to perform 10,000 random accesses of each list. Not surprisingly, the time required for the linked list is approximately three times that of the array-based list.

```
import java.util.*;

/**
 * A program to test the efficiency of random access in an ArrayList
 * versus a LinkedList.
 */
public class RandomAccess {
    /**
     * This method will perform num random accesses of the given
     * list. The return value gives the number of milliseconds
     * required to perform the lookups.
     *
     * @param the List the list to access.
     * @param num the number of times to access the list.
     * @return the number of milliseconds required to perform num
```

continued

```
     *         accesses of the list.
     */
    public static int access( List<Integer> theList, int num ) {
        Random rng = new Random();

        // Record the starting time
        long start = System.currentTimeMillis();
        // Access random locations within the list
        for ( int i = 0; i < num; i++ ) {
            Integer x = theList.get( rng.nextInt( theList.size() ) );
        }

        // Return the time required to perform the loop
        return (int)( System.currentTimeMillis() - start );
    }

    /**
     * Fill a list with num random values.
     *
     * @param theList the list to fill.
     * @param num the number of values to place in the list.
     * @param max the maximum value to place in the list.
     */
    public static void fill( List<Integer> aList, int num, int max ) {
        Random rng = new Random();

        for ( int i = 0; i < num; i++ ) {
            aList.add( rng.nextInt( max ) );
        }
    }

    /**
     * Create two lists of integers, fill them, and report the
     * time taken to randomly access them 10,000 times.
     *
     * @param args the list of command-line arguments.
     */
    public static void main( String args[] ) {
        List<Integer> aList = new ArrayList<Integer>();
        List<Integer> lList = new LinkedList<Integer>();
```

continued

```
        // Fill the lists
        fill( aList, 1000, 100 );
        fill( lList, 1000, 100 );

        // Report the time required to do 10,000 random accesses
        System.out.println( access( aList, 10000 ) );
        System.out.println( access( lList, 10000 ) );
    }
}
```

[FIGURE 9-28] Timing random access of `ArrayList` and `LinkedList` implementations

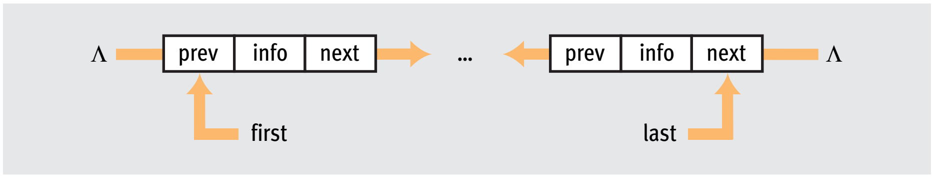
Looking back at Figure 9-27b, we can also begin to identify some `List` operations that might not work as well as the ones we just discussed—that is, operations that make insertions or deletions somewhere in the middle of the array. For example:

```
add(int index, E element); // Add element at position index
remove(int index);         // Remove the element at index
```

In both cases, we must move elements down to make space to add the new element or move elements up to eliminate the space created by the removal. Depending on the exact value of `index`, both operations could take up to O(*n*) time in the worst case. Thus, adding a new element to the front of a list or iterating through a list, removing items as you go, may work slowly in an `ArrayList` implementation. If these operations are common, you may want to consider another implementation, called `LinkedList`, which we describe in the next section. Exercise 9 at the end of the chapter asks you to use the program in Figure 9-28 to determine the performance implications of selecting an `ArrayList` versus a `LinkedList` implementation of a list.

9.4.2.2 `LinkedList`

The `LinkedList` implementation of the `List` interface is virtually identical to the doubly linked list structure discussed in Section 6.2.3.3. This implementation maintains a forward and backward pointer in each node as well as references to the list's first and last element (see Figure 9-29).



[FIGURE 9-29] Doubly linked implementation of a list

By looking at Figure 9-29, we can understand the strengths and weaknesses of this implementation. The most efficient operations in the `ArrayList` implementation—accessing and modifying by position—become much slower in a `LinkedList` implementation. For example, to retrieve the eighth item in a linked list, we either have to start at `first` and move forward seven positions, or start at `last` and move backward the correct number of positions, depending on which of the two indexes is closer to the target node. In either case, this requires in the worst case $n/2$ steps, making it $O(n)$. This compares to $O(1)$ in the `ArrayList`. Thus, if the most frequent operation performed on a list is accessing and/or modifying nodes scattered throughout the list, then `ArrayList`, with its ability to go directly to any node in constant time, is the better choice.

However, instead of accessing random nodes throughout the array, what if we want to iterate through the list, adding or deleting nodes as we traverse the structure? It is easy to traverse the linked list of Figure 9-12 by either moving forward using the methods `first()` and `next()` or backward using `last()` and `previous()`. It is also easy to delete a node or add a node to a list in $O(1)$ time once we have determined exactly where the node goes. (We showed this in Section 6.2.3.2.) Thus, if the most common operation on a list is traversing it and performing an addition or deletion operation on the nodes as we go, the `LinkedList` implementation works very well.

Another noteworthy feature of `LinkedList` is that it can grow as large as needed without ever having to be resized, like any reference-based implementation. There are no "tuning parameters" in the `LinkedList` constructors that require you to specify an initial capacity, and you do not have to make estimates about the number of elements that will be added to the list. If your lists can be of widely varying sizes, from just a few to many thousands, this may be an important consideration.

In addition to the methods in the `Collection` and `List` interfaces, the `LinkedList` implementation includes six methods not found in `ArrayList`. They are `addFirst()`, `addLast()`, `getFirst()`, `getLast()`, `removeFirst()`, and `removeLast()`. Their functions are obvious from the names, and given the existence of the `first` and `last` pointers in Figure 9-29, they can all be completed in $O(1)$ time.

Interestingly enough, none of these routines are absolutely necessary because all their operations can be performed in other ways using existing routines. For example, addFirst(e) is identical to the method call add(0,e). The method getFirst() behaves just like the call get(0). The operation removeLast() is the same as remove(size()-1). These six new methods were not added to the LinkedList implementation to extend its functionality, but as a convenience to make it easier to construct the stack and queue data structures discussed in Sections 6.3 and 6.4.

Although these additional methods may be convenient, if you decide to use them in a program, the program can only use the LinkedList implementation. If you later decide that you want to use an ArrayList, you must search through your code to find the places you used the methods specific to a LinkedList and replace them with the methods specified in the List interface. This is an example of the flexibility that you build into your code by programming to the interface. If you use only the methods provided by the List interface, your program can work with *any* class that implements the interface. On the other hand, if you write your program to use a specific implementation class, it is much more difficult to substitute a different implementation class.

Remember from those earlier discussions that you can implement both a stack and a queue using a linked list (see Figures 6-31 and 6-40, respectively). In these two data structures, we can only insert and delete items at the two ends of the list, which are called first and last in Figure 9-29. Therefore, adding, removing, and retrieving the first and last nodes in a list are identical to adding or removing elements from a stack and a queue. These equivalencies are summarized in Figure 9-30.

LinkedList ROUTINE	EQUIVALENT STACK OPERATION	EQUIVALENT QUEUE OPERATION
addFirst(o)	push(o)	
addLast(o)		enqueue(o)
o = getFirst()	o = top()	o = front()
o = getLast()		o = back()
o = removeFirst()	o = top(); pop();	e = front(); dequeue();

[FIGURE 9-30] LinkedList methods to implement a stack and a queue

For example, when you implement a stack, these additional routines allow you to think of the push operation as adding an element as the first item in the stack, addFirst(), rather than adding it to index position 0. Similarly, you can think of accessing the person at the back of a queue, getLast(), not the person at position size() - 1. It is a small difference, but Java designers felt that allowing developers to work at a higher level of abstraction was important enough to include these six additional routines in the LinkedList interface.

A special type of bidirectional iterator for `LinkedList` objects is called `ListIterator`. The class that defines this object is a private class inside the `LinkedList` implementation. Therefore, users cannot explicitly declare these types of iterators themselves. Instead, they must use the following routine in `LinkedList`:

```
ListIterator<E> listIterator(int index);
```

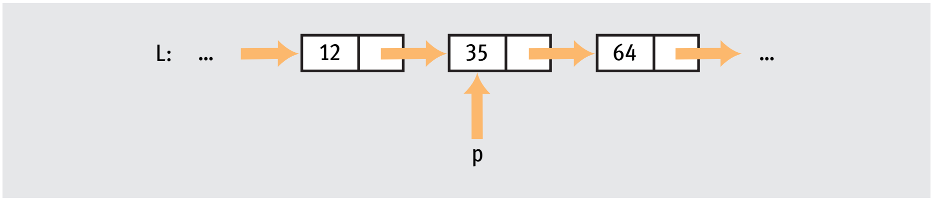
This method creates an iterator that can sequence through the nodes of the list either forward or backward. The operations on `ListIterator` objects are summarized in Figure 9-31.

The `ListIterator` *class has three other methods—*`add()`, `remove()`, *and* `set()`. *However, because they are optional and you can accomplish their functions in other ways, we do not describe them here.*

METHOD	DESCRIPTION
`boolean hasNext()`	Returns true if there are more elements in the forward direction (in other words, the iterator is not **null**) and false otherwise
`boolean hasPrevious()`	Returns true if there are more elements in the backward direction (that is, we are not positioned at the first node) and false otherwise
`E next()`	Returns the next element in the list
`E previous()`	Returns the previous element in the list
`int nextIndex()`	Returns the index of the element that would be returned by a subsequent call to `next`
`int previousIndex()`	Returns the index of the element that would be returned by a subsequent call to `previous`

[FIGURE 9-31] `ListIterator` operations

The only tricky aspect of using `ListIterators` is remembering the difference in behavior between the `next()` and `previous()` methods. Assume that a list `L` and a `ListIterator p` are in the state shown in Figure 9-32.



[FIGURE 9-32] Example of using `ListIterator` operations

The `next()` method first returns the value of the object to which it is pointing, in this case 35, and then advances to the next node, the one containing 64. The `previous()` method first retreats to the predecessor of the node it is referencing and then returns the value of that object, the integer 12. These differences in the behavior of `next()` and `previous()` mean that we must write slightly different code when traversing the list forward and backward. In the forward direction, we initialize our `ListIterator` to the first item (the one with index 0) and continue until no more nodes remain (Figure 9-33).

```
ListIterator<Integer> p = c.listIterator( 0 );
while( p.hasNext() ) {
    int x = p.next();
    // Now do something with x
}
```

[FIGURE 9-33] Traversing a `LinkedList` in the forward direction

However, when moving in reverse, we must initialize the `ListIterator` one position beyond the end of the list. If we incorrectly initialized it to the last position and then made our first call to `previous()`, we would move to the second-to-last item, return it, and ignore the last item, producing an off-by-one error. The size of the list is given by `size()`, and the indexes of the nodes go from 0 to `size() - 1`. Therefore, if we initialize our `ListIterator` to the value of `size()`, it is correctly positioned to do the traversal. This code is shown in Figure 9-34.

```
ListIterator<Integer> p = c.listIterator( size() );
while( p.hasPrevious() ) {
    int x = p.previous();
    // Now do something with x
}
```

[FIGURE 9-34] Traversing a `LinkedList` in the reverse direction

9.4.3 Maps

The `HashMap` implementation of the `Map` interface is virtually identical to the `HashSet` implementation of the `Set` interface described in Section 9.4.1.1. In fact, a `HashSet` object is an instance of a `HashMap` object in which the value field of the (*key*, *value*) tuple is ignored. Otherwise, all of our discussions regarding `HashSet` apply in exactly the same fashion to a `HashMap` implementation. This includes making sure that you properly parameterize `HashMap` using appropriate values for both of the tuning parameters, `initialCapacity` and `loadFactor`.

Similarly, the `TreeMap` implementation of the `Map` interface is identical to the `TreeSet` implementation of `Set` described in Section 9.4.1.2. Again, the only difference is that we now use the value field, whereas it was previously ignored. The `TreeMap` implementation uses the same red-black balanced tree structure described earlier. This produces O(log *n*) logarithmic behavior for insertion, deletion, and retrieval of elements to and from the map data structure.

Because the `HashMap` implementation, if properly parameterized, produces O(1) behavior for insertion and retrieval, we ask the same question as before: Why would we ever want to use a `TreeMap`? The answer is the same: A `TreeMap` implementation guarantees that the map can be efficiently traversed in sorted sequence according to the "natural order" of the class of the key field. (See Section 9.3.5.1 for a discussion of natural ordering of elements and the behavior of the two methods `compareTo()` and `equals()`.)

This ability to do a sorted traversal of keys is critical for the `SortedMap` and `SortedSet` interfaces discussed earlier in this chapter, because the iterator returned by the `iterator()` method in `Collection` must traverse a `SortedMap` or `SortedSet` in key order. This can be done efficiently when using a `TreeMap` or `TreeSet` implementation, but it is extremely slow with the `HashMap` and `HashSet`. In addition, when applied to a sorted collection (either `SortedSet` or `SortedMap`), the arrays returned by the `toArray()` methods must store the keys, values, or entries in key order.

Because you need to provide ordered traversals and ordered array storage to the set of keys or the map of (*key*, *value*) tuples, the only implementations of `SortedMap` and `SortedSet` provided by the Java Collection Framework are `TreeMap` and `TreeSet`, respectively.

9.5 Algorithms

The final part of the Java Collection Framework we discuss in this chapter is the **algorithms** that operate on the different types of collections in the framework. The algorithms take the form of **static** class methods, and are found in the `Collections` and `Arrays` classes. The `Collections` class provides algorithms that are designed to operate on arbitrary collections, while the algorithms in the `Arrays` class are designed to operate on standard Java arrays. We begin with the algorithms in the `Collections` class.

Most methods in the `Collections` class work on lists, but a few methods operate on arbitrary collections. Many of the algorithmic methods listed in Figure 9-35 require the collection on which they work to be ordered. For example, you can only swap elements in an ordered collection because elements do not have a position in an unordered collection, and the concept of swapping is meaningless.

METHOD	DESCRIPTION
`binarySearch();`	Searches a list for the specified value
`sort();`	Sorts a list
`reverse();`	Reverses the contents of a list
`fill();`	Fills a list with the specified value
`copy();`	Copies a list
`shuffle();`	Randomly shuffles the contents of a list
`max();`	Returns the maximum value in a collection
`min();`	Returns the minimum value in the collection
`replaceAll();`	Replaces all occurrences of a value in a list with a different value
`rotate();`	Rotates the contents of a list
`swap();`	Swaps the elements in a list

[FIGURE 9-35] Some algorithms in the Java Collection Framework

The `Collections` class uses the binary search algorithm (discussed in Section 5.3.3) to search for specific elements in a list. Thus, you must sort the list before invoking the `binarySearch()` method. The method has two forms; the first, which takes only a list and a key as parameters, assumes that elements in the list are sorted into ascending order according to their natural order. In other words, it assumes that the elements in the list implement the `Comparable` interface. The second form also includes a `Comparator` parameter `c`, and can be used for lists whose elements do not have a natural order or for lists that require some other ordering for purposes of the search. A positive return value from `binarySearch()` indicates that the target was found in the list; the return value also gives the index position where the element was found. A negative return value indicates that the target was not found in the list.

The `sort()` method, like the `binarySearch()` method, has two forms: one that uses the natural order of the elements in the list and a second that specifies a comparator to impose an ordering on the list's elements. The sort method repositions the elements so that they appear in ascending order according to either the natural order of the list's elements or to the comparator. The sort method uses the $O(n \log_2 n)$ merge sort algorithm that we discussed and analyzed in Section 5.5.

An important feature of a merge sort is that it is **stable**. This means that it does not reorder equal elements. If two elements in the list are equal, they appear in the same order in the sorted list. This fact is useful when you have to sort a list by two different criteria. For example, consider the `Dog` class in Figure 9-19. If we wanted to sort a list of dogs to be ordered by breed and then by name, we could sort the list once by name and then sort it a second time by breed. All dogs whose breeds are the same would appear in alphabetical order by name.

The following code shows how to use the `sort()` and `binarySearch()` methods to find a value in a list. Note that you must sort the list before calling the binary search because the list itself is not ordered.

```
Collections.sort( dogs );
if ( Collections.binarySearch( dogs, target ) ) < 0 ) {
    System.out.println( target + " is not in the list" );
}
```

You can use the `max()` and `min()` methods to find the maximum and minimum values in an arbitrary collection. These are the only methods in the `Collections` class that work on a collection. All they require is the ability to iterate over the collection, which is something that all collections provide. By now, you should have a good feel for the methods in the `Collections` class, and you should not be surprised that there are two forms of `max()` and `min()`. One form assumes that the elements in the list are comparable, and the second takes a comparator that you use to order the elements.

If you invoke a form of a `Collections` method that expects its elements to be comparable, but they are not, an error results. The code that implements the first form of the `max()` method uses an `Iterator` to individually examine each element in the collection.

The `next()` method of the iterator returns an `Object` that, in this situation, is cast to `Comparable`. However, if the object does not implement the `Comparable` interface, then a `ClassCastException` is thrown. This holds true for the `binarySearch()` and `sort()` methods as well.

The remaining methods in Figure 9-35 provide algorithms to manipulate the data stored in a `List` in a variety of ways. The shuffle algorithm is the opposite of a sort: It randomizes the position of elements within a list. The following code shows how to create a list called `aList` that contains the integer values from 0 to 99 in some random order:

```
List<Integer> aList = new ArrayList<Integer>();
for ( int i = 0; i < 100; i++ ) {
    aList.add( i );
}
Collections.shuffle( aList );
```

The `Collections` class also provides a number of **wrapper methods** that can change the functionality of a collection in some interesting and useful ways. The only wrapper methods we present here are called **unmodifiable wrappers**, which are more commonly termed **read-only wrappers**.

These methods are easy to explain—they basically allow a user to access and traverse a collection, but users cannot change it in any way. Thus, operations like `get()`, `next()`, `previous()`, `size()`, and `isEmpty()`, which do not alter the elements of the collection, all function in exactly the same way we have described. However, operations like `add()`, `remove()`, and `set()` are explicitly prohibited. If you have declared a collection to be read-only, then any attempt to modify the collection is intercepted by the system, which throws an `UnsupportedOperationException` and terminates your program. (You learn how to deal with exception objects in the following chapter.)

You create a read-only collection in two steps. First, create a regular collection in exactly the way we explained it in this chapter. This could be a `Collection`, `Set`, `Map`, `List`, `SortedSet`, or `SortedMap`. Then you "wrap" the collection inside a read-only collection using one of the following six **static** methods that correspond to each of the regular collection types:

```
public static Collection unmodifiableCollection(Collection c);
public static Set unmodifiableSet(Set s);
public static List unmodifiableList(List l);
public static Map unmodifiableMap(Map m);
public static SortedSet unmodifiableSortedSet(SortedSet s);
public static SortedMap unmodifiableSortedMap(SortedMap m);
```

The collection returned by these methods contains exactly the same elements in the initial collection, but is a read-only object that cannot be modified. Because the read-only collection wraps the original collection, any changes to the original collection are seen by the read-only version. However, you cannot change the original collection given a reference to the read-only wrapper. One advantage of these read-only collections is that you can create two distinct classes of users. For example, consider the following declaration:

```
Set s; // The original set collection
Set ums; // The unmodifiable set
// Build the set s (code omitted)
ums = unmodifiableSet( s ); // Create a read-only set ums
```

Some "first-class" users can do anything they want. For these users, you pass a reference to `s`, the original collection. They can look at `s` as well as change `s`. However, second-class users can only "look but not touch." For these users, you pass a reference to `ums`. Any attempt by these users to modify `ums` is trapped and recognized as an error. Any changes a first-class user makes to the collection are seen by the second-class users.

Another interesting application of these read-only collections is in the software development process. Assume that your code initially builds a collection. Then, for the rest of the program, all you do is work with those elements, but you never change them. One example might be a dictionary of words. Once you build it, you only want to look up words; you don't want to remove words or modify definitions. One way to ensure that your program behaves correctly is to make your dictionary a read-only structure, as follows:

```
// The contents of rwDictionary may be changed
List<String> rwDictionary = new ArrayList<String>();

// Build the dictionary

// The contents of roDictionary cannot be changed
List<String> roDictionary = Collections.unmodifiableList(d);
```

Now we discard the reference to the original dictionary, `d`, and keep only the read-only reference, called `dictionary`. All coding is done in terms of the read-only object `dictionary`, and the program cannot change this collection. You can be sure of avoiding bugs in your software that would be caused by unexpected and incorrect changes to your dictionary data structure.

The `Arrays` class provides methods that allow you to convert an array to a list, search the array, sort the array, determine if two arrays are equal, and fill the array with a specific value. The methods in the `Arrays` class behave in the same way as those we just described in the `Collections` class. The biggest difference you can see in the `Arrays` class is that it provides overloaded methods that work with arrays of any type.

9.6 Summary

Virtually all assignments in a first computer science course ask students (either individually or as part of a team) to build a program from scratch. All code, with the exception of such minor library routines as `abs()` or `sqrt()`, is original and designed, written, and tested by the students themselves. This is necessary because the courses focus on teaching fundamental concepts of programming languages and implementation. It is essential that you understand these concepts.

However, this approach does not model how programs are written in the real world. Building an application from scratch is slow, extremely expensive, and highly prone to errors. Programmers try to reuse as much existing code as they can, either from the standard libraries provided by the language, software in the public domain, or from their own private collection of helpful routines. The policy for software development in the real world is not "how do I build it?" but "where can I find it?"

The discussion of the Java Collection Framework in this chapter should have made this point clear. Although Chapters 6, 7, and 8 discussed fundamental concepts in algorithms, complexity, and data structures, along with implementations and massive amounts of original code to demonstrate the concepts being discussed, Chapter 9 is more realistic about how modern software developers actually work with these data structures.

The Java Collection Framework provides interfaces and implementations for virtually all of the structures we have discussed, so you should be familiar and comfortable with its ideas. When you have a problem to solve and a program to write, you don't have to start at the beginning. Instead, you scan the available libraries and select the most appropriate and efficient structures to solve your problem. Not only does this keep costs down, it goes a long way toward ensuring the correctness of the completed software.

GRACE MURRAY HOPPER

Grace Murray Hopper was one of the most influential computer scientists of the 20th century. She was born in New York City on December 9, 1906, graduated from Vassar College in 1928, and received a PhD in Mathematics from Yale University in 1934. She was a member of the Vassar faculty from 1931 to 1943, and then joined the Naval Reserve. She was commissioned as a lieutenant in 1944 and was assigned to the Bureau of Ordnance, where she helped develop the electronic computer. She was promoted to captain in 1973, commodore in 1983, and rear admiral in 1985.

Admiral Hopper was consistently at the forefront of progress in computers and programming languages, and she worked with some of the earliest computing systems. While working for the Navy, she programmed the Mark I, Mark II, and Mark III computers, and while working for the Eckert-Mauchly Computer Corporation, she did pioneering work on the UNIVAC I, the first large-scale electronic digital computer.

Her most significant contributions to the field were in the area of programming. Admiral Hopper encouraged programmers to collect and share common portions of programs. She understood that sharing code reduced errors, decreased development time, and made the programming process less tedious. One of her most notable achievements was the development of the COBOL programming language. In later years, Admiral Hopper was often referred to as Mother COBOL or Grandma COBOL for her part in creating what is still one of the world's most widely used programming languages.

Hopper had a colorful personality, a gift for speaking, and a talent for education. The clock in her office ran counterclockwise to remind her that every approach has an alternative. Hopper was famous for her teaching skills and her use of analogies. For example, when asked why satellite communication took so long, she would hand out wires that were about a foot long—light can travel about one foot in a nanosecond. She would then point to a spool of wire about 1000 feet long, explain that it represented a microsecond, and point out that it took several microseconds for a signal to reach a satellite.

Late in her career, when asked which of her accomplishments she was most proud of, she would often reply, "All the young people I have trained over the years." Admiral Hopper remained active in industry and education until her death in 1992. In recognition of her achievements, the Navy named a ship in her honor and the Association for Computing Machinery created the Grace Murray Hopper Award in 1971. The award is given to a young computer professional who makes a significant technical or service contribution. The programming languages and libraries you use today to develop software were all influenced by the work of Admiral Hopper.

EXERCISES

1 Using the Java Collection Framework, write the following method:

```
public int countKey(List L, Object key);
```

The method iterates through the list `L`, using a `ListIterator`, and returns the total number of times that the object `key` occurs in the list.

2 Using the Java Collection Framework, write the following method:

```
public void deleteKey(List L, Object key);
```

The method iterates through the list `L`, using a `ListIterator`, and deletes every occurrence of the object `key` in the list.

3 Referring to the `BookShelf` class in Figure 9-16, write the following method:

```
public Collection getAllISBN();
```

This method returns a `Collection` that contains the ISBN of every book in the `BookShelf`.

4 Assume that the `Name` class of Figure 9-18 includes a first name, last name, and middle name. In addition, an accessor method, `getMiddle()`, returns the middle name. Rewrite the `compareTo()` method of `Name` to have the following rules for comparing two names:

a A name is greater than another name if its last name is greater (in an alphabetical sense).

b If the last names are the same, then the name with the greater first name is greater.

c If both the first and last names are the same, then the middle name with the greater middle name is greater.

5 Write a `Comparator` for the `Dog` class of Figure 9-19 that implements comparisons in the following way:

- a Female dogs are greater than male dogs.
- b For two dogs of the same gender, order the dogs by their name.
- c For dogs of the same gender and name, order the dogs by their breed.

6 Assume that we have a `SortedSet`, called `serialNumbers`, that contains an ordered list of the five-character serial numbers described in Section 9.3.5.2. The first character of the serial number is a letter A, B, ... that identifies a particular vendor. The last four digits 0000, 0001, ... identify a piece of equipment from the vendor. Assume that our 13 current vendors are assigned the letters A to M. Write a method that produces a table listing the number of pieces of equipment provided by each vendor.

7 Using the built-in `hashCode()` method inherited by all objects from `Object`, perform the following computations:

```
int a[50];
for (int k = 0; k < 10000; k++) {
  // Generate a random integer n and convert n
  // to an Integer object nObj using the Integer class
  // This code is not shown
  i = ( nObj.hashCode() ) % 50;
  a[ i ]++;
}
```

Your function should generate 10,000 values, each in the range 0–49. If the hashing function `hashCode()` is working well, it should scatter these 10,000 values evenly, producing about 200 hits to each of the 50 buckets in the array `a`.

Run this test program and examine how many values hashed to each of the 50 locations. Use the data to determine whether the built-in hashing function seems to be working well. (If you know something about statistics, you can do this analysis formally using a chi-square goodness of fit test. Otherwise, do an eyeball analysis.)

8 Construct a `HashSet` implementation of a `Set` using a range of different values for the `initialCapacity` and `loadFactor` parameters. Measure exactly how long it takes to add 10,000 elements to the set and delete 10,000 from the set. Collect these times and graph them as a function of the different values of these two parameters. Discuss the effect that the parameter values have on the time it takes to insert and delete elements using a `HashSet` implementation.

9 Build a `List` data structure using both an `ArrayList` and a `LinkedList` implementation. Store 10,000 elements into the `List` (the type of the elements does not matter), and time how long it takes to do each of the following three operations:

a Do 1000 retrievals of a randomly selected position in the `List`.

b Insert 1000 new elements into the `List` in a random location.

c Delete 1000 randomly selected elements from the `List`.

Use the data you collect to discuss the performance implications of these two different implementations.

10 Write a program that reads the contents of a text file and produces a report that lists the 10 most common and 10 least common words. Write the program so that the name of the file to process appears on the command line.

To have some fun with your program after writing it, look on the Web for Project Gutenberg (*http://promo.net/pg/*), a project that converts classic literature to electronic form.

11 List two reasons you might want to use a comparator in a Java program.

12 What output is generated when the following program is executed?

```
import java.util.*;

public class Program1 {
  public static void main( String args[] ) {
    List<Book> lib = new ArrayList<Book>();
    lib.add(new Book("Core Java", "Horstmann", 630));
    lib.add(new Book("Unix Power Tools", "Peek", 1127));
    lib.add(new Book("Java", "Buster", 1995));
    lib.add(new Book("Java", "Grumpy", 423));
    list( lib );
    System.out.println();
    Collections.sort( lib, new ByTitleAndAuthor());
    list( lib );
  }

  public static void list( List l ) {
    for ( Book b : l ) {
      System.out.println( b );
    }
  }
}
```

13 You are designing an application that requires elements in a collection to be maintained in the sequence in which they are added. In addition, you know that elements will often be added and removed from the middle of the collection. Select the most appropriate implementation class and justify your selection.

14 In creating an object such as a `Set` or a `Map`, we choose an actual implementation, such as a `HashSet` or a `HashMap`. In declaring this object, we prefer to use an interface, such as `Set` or `Map`. Explain why and list the benefits of doing this.

15 The `LinkedList` class provides several methods that are not defined in the `List` interface. Why should you not use these "special" linked list methods in a program?

16 Which sequence of digits does the following program print?

```
public class Lists {
  public static void main( String args[] ) {
    List<String> list = new ArrayList<String>();

    list.add( "1" );
    list.add( "2" );
    list.add( 1, "3" );

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

```
        list.addAll( list2 );
        list2 = list.subList( 2, 5 );
        list2.clear();

        System.out.println( list );
    }
}
```

17 Normally, a telephone book has alphabetical entries listed by the owner of the telephone number. This makes it almost impossible, given only a number, to find the name of its owner. Write a `PhoneBook` class that manages a phone directory—you should be able to add, change, and delete directory entries. The class must provide two search methods, `getByName()` and `getByNumber()`, that retrieve the directory information for a person, given the person's name or number. Both methods must perform their searches in O(1) time.

18 Design and implement a class that simulates the operation of a cash register. Your class must include the following methods:

a `purchase()`, which takes as parameters the universal product code (UPC), description, and cost of an item that is being purchased

b `coupon()`, which takes the UPC of the item a coupon applies to and the amount of money the coupon is worth

c `printReceipt()`, which prints a receipt for the purchased items and redeemed coupons. Items must be listed in the same order as the customer purchased them, and coupons must be listed in order based on the UPC of their applicable item. The receipt must show the total cost of all the items purchased, the total value of the coupons, and the total owed by the customer.

The register should handle coupons as follows:

a Duplicate coupons (in other words, two coupons with the same UPC code) are not accepted. The cash register accepts only the first coupon.

b A coupon is not accepted unless the corresponding item (an item with the same UPC code) has been purchased.

c If the value of the coupon is more than the sale price of its applicable item, the coupon's value is reduced to match the price of the item.

19 Write a class that implements a dictionary and provides methods to add words to it. Include another method named `spellCheck()` that takes a string as an argument and returns a list that determines if the word is in the dictionary. If it is, the list will be empty; otherwise, a sorted list will appear with possible spellings for the word. Write your class so that the `spellCheck()` method runs as quickly as possible.

20 Consider the following two lines of code that declare and create an instance of an `ArrayList`:

```
List<Automobile> cars = new ArrayList<Automobile>();
ArrayList<Automobile> cars = new ArrayList<Automobile>();
```

Which line represents the "correct" way to work with a list in a Java program? Explain your answer.

21 Write a class named `Card` that represents a card in a standard bridge deck. Using the `Card` class, write a class named `BridgeDeck` that represents a standard bridge deck. Your `BridgeDeck` class should provide methods to shuffle and deal cards. Your deck should only deal each card once, and it eventually runs out of cards.

22 A company has asked you to write a security system. The company has issued identification cards that contain RFID (radio frequency identification) tags to each of its employees. As an employee enters or leaves the building, a scanner automatically reads the card and sends the employee's unique identification number to your system. At the end of the day, your system must print two reports: one that lists by employee identification number the amount of time each employee spent in the building (an employee may enter and leave several times each day), and a second report that lists the times the employees entered and left the building, sorted by time. You must build the system so that it uses one `Map` to track who is in the building.

23 Modify the `BinaryTree` interface from Chapter 7 so that it has three additional methods: `inorderIterator()`, `preorderIterator()`, and `postorderIterator()`. When invoked, each method should return an object that implements the `java.util.Iterator` interface and iterates over the tree in the order specified. Modify the reference-based tree implementation in Chapter 7 to provide implementations for these methods.

24 Write a program that measures the amount of time it takes to sort collections of various sizes using the `Collections.sort()` method. Then see if you can do better. Write your own sort algorithm using every trick you can think of to make your sort as fast as possible (keep track of how long it takes to write your sort method). Can you beat the sort routine provided in the Java API? How much effort did it take? Was it worth it?

25 The Standard Template Library (STL) provides a number of classes that C++ programmers can use. Find out what collections are provided by the STL and explain how they compare to the Java Collection framework.

CHALLENGE WORK EXERCISES

1 Design a `SortedList` interface for the Java Collection Framework. A `SortedList` should implement `java.util.List` and provide methods like those found in `SortedSet` and `SortedMap`, which take advantage of the fact that the list is sorted. Write two classes that implement your `SortedList` interface using an array-based list and a linked list.

Write a program that tests your `SortedList` implementations and measures their performance. Does it make sense to implement a sorted list using an array?

2 The `TreeSet` class uses a red-black tree to avoid building unbalanced trees. An AVL tree is similar to a red-black tree, in that it reorganizes itself occasionally to maintain balance. Find a description of AVL trees, and write a class named `AVLTreeSet` that implements the `Set` interface using AVL trees.

3 Using classes from the Java Collection Framework, write an implementation of the `Graph` interface in Figure 8-18.