CHAPTER 15

THE JAVA COLLECTIONS FRAMEWORK

CHAPTER GOALS

To learn how to use the collection classes supplied in the Java library

To use iterators to traverse collections

To choose appropriate collections for solving programming problems

To study applications of stacks and queues

CHAPTER CONTENTS

If you want to write a program that collects objects (such as the stamps to the left), you have a number of choices. Of course, you can use an array list, but computer scientists have invented other mechanisms that may be better suited for the task. In this chapter, we introduce the collection classes and interfaces that the Java library offers. You will learn how to use the Java collection classes, and how to choose the most appropriate collection type for a problem.

15.1 An Overview of the Collections Framework

A collection groups together elements and allows them to be retrieved later.

When you need to organize multiple objects in your program, you can place them into a **collection**. The `ArrayList` class that was introduced in Chapter 6 is one of many collection classes that the standard Java library supplies. In this chapter, you will learn about the Java *collections framework*, a hierarchy of interface types and classes for collecting objects. Each interface type is implemented by one or more classes (see Figure 1).

At the root of the hierarchy is the `Collection` interface. That interface has methods for adding and removing elements, and so on. Table 1 on page W672 shows all the methods. Because all collections implement this interface, its methods are available for all collection classes. For example, the `size` method reports the number of elements in *any* collection.

The `List` interface describes an important category of collections. In Java, a *list* is a collection that remembers the order of its elements (see Figure 2). The `ArrayList` class implements the `List` interface. The Java library supplies another class, `LinkedList`, that also implements the `List` interface. Unlike an array list, a linked list allows speedy insertion and removal of elements in the middle of the list. We will discuss that class in the next section.

A list is a collection that remembers the order of its elements.

You use a list whenever you want to retain the order that you established. For example, on your bookshelf, you may order books by topic. A list is an appropriate data structure for such a collection because the ordering matters to you.

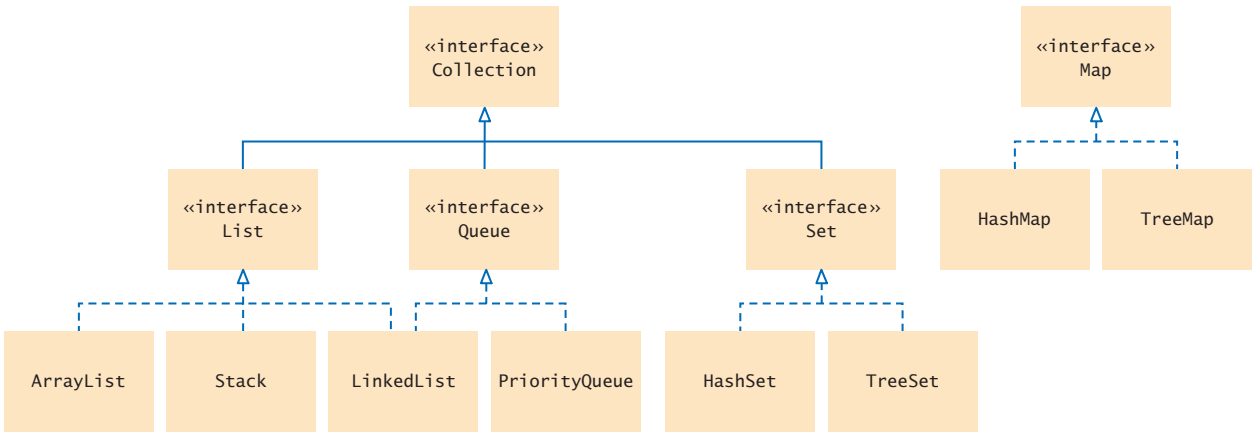


Figure 1 Interfaces and Classes in the Java Collections Framework

Figure 2 A List of Books

Figure 3 A Set of Books

Figure 4 A Stack of Books

A set is an unordered collection of unique elements.

However, in many applications, you don't really care about the order of the elements in a collection. Consider a mail-order dealer of books. Without customers browsing the shelves, there is no need to order books by topic. Such a collection without an intrinsic order is called a **set**—see Figure 3.

Because a set does not track the order of the elements, it can arrange them in a way that speeds up the operations of finding, adding, and removing elements. Computer scientists have invented mechanisms for this purpose. The Java library provides classes that are based on two such mechanisms (called *hash tables* and *binary search trees*). You will learn in this chapter how to choose between them.

Another way of gaining efficiency in a collection is to reduce the number of operations. A **stack** remembers the order of its elements, but it does not allow you to insert elements in every position. You can add and remove elements only at the top—see Figure 4.

In a **queue**, you add items to one end (the tail) and remove them from the other end (the head). For example, you could keep a queue of books, adding required reading at the tail and taking a book from the head whenever you have time to read another one. We will discuss stacks and queues in Section 15.5.

A map keeps associations between key and value objects.

Finally, a **map** manages associations between *keys* and *values*. Every key in the map has an associated value. The map stores the keys, values, and the associations between them. For an example, consider a library that puts a bar code on each book.

The program used to check books in and out needs to look up the book associated with each bar code. A map associating bar codes with books can solve this problem—see Figure 5. We will discuss maps in Section 15.4.

Figure 5 A Map from Bar Codes to Books

ONLINE EXAMPLE

A sample program that demonstrates several collection classes.

Table 1 The Methods of the `Collection` Interface

`Collection<String> coll = new ArrayList<String>();`	The `ArrayList` class implements the `Collection` interface.
`coll = new TreeSet<String>();`	The `TreeSet` class (Section 15.3) also implements the `Collection` interface.
`int n = coll.size();`	Gets the size of the collection. `n` is now `0`.
`coll.add("Harry");` `coll.add("Sally");`	Adds elements to the collection.
`String s = coll.toString();`	Returns a string with all elements in the collection. `s` is now `"[Harry, Sally]"`
`System.out.println(coll);`	Invokes the `toString` method and prints `[Harry, Sally]`.
`coll.remove("Harry");` `boolean b = coll.remove("Tom");`	Removes an element from the collection, returning `false` if the element is not present. `b` is false.
`b = coll.contains("Sally");`	Checks whether this collection contains a given element. `b` is now `true`.

`for (String s : coll)`
`{`
`  System.out.println(s);`
`}`
	You can use the "for each" loop with any collection. This loop prints the elements on separate lines.

`Iterator<String> iter = coll.iterator()`	You use an iterator for visiting the elements in the collection (see Section 15.2.3).

1. A gradebook application stores a collection of quizzes. Should it use a list or a set?
2. A student information system stores a collection of student records for a university. Should it use a list or a set?
3. Why is a queue of books a better choice than a stack for organizing your required reading?
4. As you can see from Figure 1, the Java collections framework does not consider a map a collection. Give a reason for this decision.

Practice It Now you can try these exercises at the end of the chapter: R15.1, R15.2, R15.3.

15.2 Linked Lists

A **linked list** is a data structure used for collecting a sequence of objects that allows efficient addition and removal of elements in the middle of the sequence. In the following sections, you will learn how a linked list manages its elements and how you can use linked lists in your programs.

15.2.1 The Structure of Linked Lists

Each node in a linked list is connected to the neighboring nodes.

To understand the inefficiency of arrays and the need for a more efficient data structure, imagine a program that maintains a sequence of employee names. If an employee leaves the company, the name must be removed. In an array, the hole in the sequence needs to be closed up by moving all objects that come after it. Conversely, suppose an employee is added in the middle of the sequence. Then all names following the new hire must be moved toward the end. Moving a large number of elements can involve a substantial amount of processing time. A linked list structure avoids this movement.

A linked list consists of a number of nodes, each of which has a reference to the next node.

A linked list uses a sequence of *nodes*. A node is an object that stores an element and references to the neighboring nodes in the sequence (see Figure 6).

Figure 6
A Linked List

When you insert a new node into a linked list, only the neighboring node references need to be updated (see Figure 7).

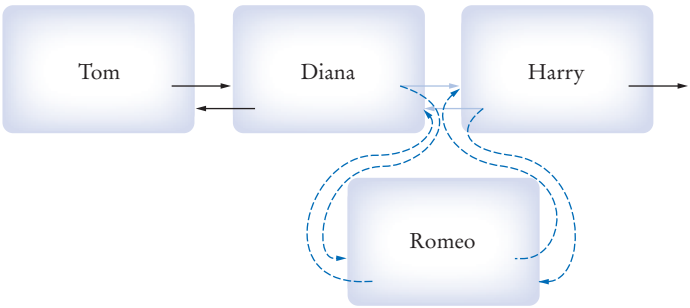


Figure 7
Inserting a Node into a Linked List

The same is true when you remove a node (see Figure 8). What's the catch? Linked lists allow speedy insertion and removal, but element access can be slow.

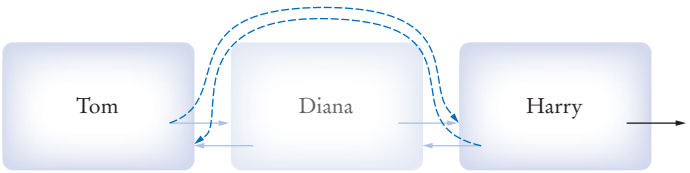


Figure 8
Removing a Node from a Linked List

Adding and removing elements at a given location in a linked list is efficient.

Visiting the elements of a linked list in sequential order is efficient, but random access is not.

For example, suppose you want to locate the fifth element. You must first traverse the first four. This is a problem if you need to access the elements in arbitrary order. The term "random access" is used in computer science to describe an access pattern in which elements are accessed in arbitrary (not necessarily random) order. In contrast, sequential access visits the elements in sequence.

Of course, if you mostly visit all elements in sequence (for example, to display or print the elements), the inefficiency of random access is not a problem. You use linked lists when you are concerned about the efficiency of inserting or removing elements and you rarely need element access in random order.

15.2.2 The `LinkedList` Class of the Java Collections Framework

The Java library provides a `LinkedList` class in the `java.util` package. It is a **generic class**, just like the `ArrayList` class. That is, you specify the type of the list elements in angle brackets, such as `LinkedList<String>` or `LinkedList<Employee>`.

Table 2 shows important methods of the `LinkedList` class. (Remember that the `LinkedList` class also inherits the methods of the `Collection` interface shown in Table 1.)

As you can see from Table 2, there are methods for accessing the beginning and the end of the list directly. However, to visit the other elements, you need a list **iterator**. We discuss iterators next.

Table 2 Working with Linked Lists

`LinkedList<String> list = new LinkedList<String>();`	An empty list.
`list.addLast("Harry");`	Adds an element to the end of the list. Same as `add`.
`list.addFirst("Sally");`	Adds an element to the beginning of the list. `list` is now `[Sally, Harry]`.
`list.getFirst();`	Gets the element stored at the beginning of the list; here `"Sally"`.
`list.getLast();`	Gets the element stored at the end of the list; here `"Harry"`.
`String removed = list.removeFirst();`	Removes the first element of the list and returns it. `removed` is `"Sally"` and `list` is `[Harry]`. Use `removeLast` to remove the last element.
`ListIterator<String> iter = list.listIterator()`	Provides an iterator for visiting all list elements (see Table 3 on page W676).

15.2.3 List Iterators

An iterator encapsulates a position anywhere inside the linked list. Conceptually, you should think of the iterator as pointing between two elements, just as the cursor

You use a list iterator to access elements inside a linked list.

in a word processor points between two characters (see Figure 9). In the conceptual view, think of each element as being like a letter in a word processor, and think of the iterator as being like the blinking cursor between letters.

You obtain a list iterator with the `listIterator` method of the `LinkedList` class:

```
LinkedList<String> employeeNames = . . .;
ListIterator<String> iterator = employeeNames.listIterator();
```

Note that the iterator class is also a generic type. A `ListIterator<String>` iterates through a list of strings; a `ListIterator<Book>` visits the elements in a `LinkedList<Book>`.

Initially, the iterator points before the first element. You can move the iterator position with the `next` method:

```
iterator.next();
```

The `next` method throws a `NoSuchElementException` if you are already past the end of the list. You should always call the iterator's `hasNext` method before calling `next`—it returns `true` if there is a next element.

```
if (iterator.hasNext())
{
   iterator.next();
}
```

The `next` method returns the element that the iterator is passing. When you use a `ListIterator<String>`, the return type of the `next` method is `String`. In general, the return type of the `next` method matches the list iterator's type parameter (which reflects the type of the elements in the list).

You traverse all elements in a linked list of strings with the following loop:

```
while (iterator.hasNext())
{
   String name = iterator.next();
   Do something with name
}
```

As a shorthand, if your loop simply visits all elements of the linked list, you can use the "for each" loop:

```
for (String name : employeeNames)
{
   Do something with name
}
```

Then you don't have to worry about iterators at all. Behind the scenes, the `for` loop uses an iterator to visit all list elements.

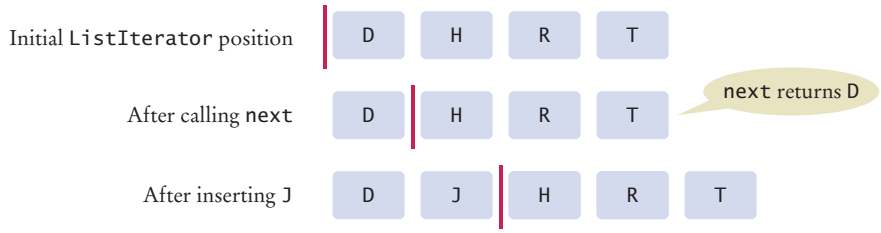


Figure 9 A Conceptual View of the List Iterator

The nodes of the `LinkedList` class store two links: one to the next element and one to the previous one. Such a list is called a **doubly-linked list**. You can use the `previous` and `hasPrevious` methods of the `ListIterator` interface to move the iterator position backward.

The `add` method adds an object after the iterator, then moves the iterator position past the new element.

```
iterator.add("Juliet");
```

You can visualize insertion to be like typing text in a word processor. Each character is inserted after the cursor, then the cursor moves past the inserted character (see Figure 9). Most people never pay much attention to this—you may want to try it out and watch carefully how your word processor inserts characters.

The `remove` method removes the object that was returned by the last call to `next` or `previous`. For example, this loop removes all names that fulfill a certain condition:

```
while (iterator.hasNext())
{
   String name = iterator.next();
   if (condition is fulfilled for name)
   {
      iterator.remove();
   }
}
```

You have to be careful when calling `remove`. It can be called only *once* after calling `next` or `previous`, and you cannot call it immediately after a call to `add`. If you call the method improperly, it throws an `IllegalStateException`.

Table 3 summarizes the methods of the `ListIterator` interface. The `ListIterator` interface extends a more general `Iterator` interface that is suitable for arbitrary collections, not just lists. The table indicates which methods are specific to list iterators.

Following is a sample program that inserts strings into a list and then iterates through the list, adding and removing elements. Finally, the entire list is printed. The comments indicate the iterator position.

Table 3 Methods of the `Iterator` and `ListIterator` Interfaces

`String s = iter.next();`	Assume that `iter` points to the beginning of the list `[Sally]` before calling `next`. After the call, `s` is `"Sally"` and the iterator points to the end.
`iter.previous();` `iter.set("Juliet");`	The `set` method updates the last element returned by `next` or `previous`. The list is now `[Juliet]`.
`iter.hasNext()`	Returns `false` because the iterator is at the end of the collection.

`if (iter.hasPrevious())`
`{`
`  s = iter.previous();`
`}`
	`hasPrevious` returns `true` because the iterator is not at the beginning of the list. `previous` and `hasPrevious` are `ListIterator` methods.

`iter.add("Diana");`	Adds an element before the iterator position (`ListIterator` only). The list is now `[Diana, Juliet]`.
`iter.next();` `iter.remove();`	`remove` removes the last element returned by `next` or `previous`. The list is now `[Diana]`.

section_2/ListDemo.java

```
1  import java.util.LinkedList;
2  import java.util.ListIterator;
3
4  /**
5     This program demonstrates the LinkedList class.
6  */
7  public class ListDemo
8  {
9     public static void main(String[] args)
10    {
11       LinkedList<String> staff = new LinkedList<String>();
12       staff.addLast("Diana");
13       staff.addLast("Harry");
14       staff.addLast("Romeo");
15       staff.addLast("Tom");
16
17       // | in the comments indicates the iterator position
18
19       ListIterator<String> iterator = staff.listIterator(); // |DHRT
20       iterator.next(); // D|HRT
21       iterator.next(); // DH|RT
22
23       // Add more elements after second element
24
25       iterator.add("Juliet"); // DHJ|RT
26       iterator.add("Nina"); // DHJN|RT
27
28       iterator.next(); // DHJNR|T
29
30       // Remove last traversed element
31
32       iterator.remove(); // DHJN|T
33
34       // Print all elements
35
36       System.out.println(staff);
37       System.out.println("Expected: [Diana, Harry, Juliet, Nina, Tom]");
38    }
39 }
```

Program Run

```
[Diana, Harry, Juliet, Nina, Tom]
Expected: [Diana, Harry, Juliet, Nina, Tom]
```

5. Do linked lists take more storage space than arrays of the same size?
6. Why don't we need iterators with arrays?
7. Suppose the list `lst` contains elements `"A"`, `"B"`, `"C"`, and `"D"`. Draw the contents of the list and the iterator position for the following operations:

```
ListIterator<String> iter = letters.iterator();
iter.next();
iter.next();
iter.remove();
iter.next();
iter.add("E");
```

```
iter.next();
iter.add("F");
```

8. Write a loop that removes all strings with length less than four from a linked list of strings called `words`.
9. Write a loop that prints every second element of a linked list of strings called `words`.

Practice It Now you can try these exercises at the end of the chapter: R15.4, R15.7, P15.1.

Random Fact 15.1 Standardization

You encounter the benefits of standardization every day. When you buy a light bulb, you can be assured that it fits the socket without having to measure the socket at home and the light bulb in the store. In fact, you may have experienced how painful the lack of standards can be if you have ever purchased a flashlight with nonstandard bulbs. Replacement bulbs for such a flashlight can be difficult and expensive to obtain.

Programmers have a similar desire for standardization. Consider the important goal of platform independence for Java programs. After you compile a Java program into class files, you can execute the class files on any computer that has a Java virtual machine. For this to work, the behavior of the virtual machine has to be strictly defined. If all virtual machines don't behave exactly the same way, then the slogan of "write once, run anywhere" turns into "write once, debug everywhere". In order for multiple implementors to create compatible virtual machines, the virtual machine needed to be standardized. That is, someone needed to create a definition of the virtual machine and its expected behavior.

Who creates standards? Some of the most successful standards have been created by volunteer groups such as the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C). The IETF standardizes protocols used in the Internet, such as the protocol for exchanging e-mail messages. The W3C standardizes the Hypertext Markup Language (HTML), the format for web pages. These standards have been instrumental in the creation of the World Wide Web as an open platform that is not controlled by any one company.

Many programming languages, such as C++ and Scheme, have been standardized by independent standards organizations, such as the American National Standards Institute (ANSI) and the International Organization for Standardization—called ISO for short (not an acronym; see `http://www.iso.org/iso/about/discover-iso_isos-name.htm`). ANSI and ISO are associations of industry professionals who develop standards for everything from car tires to credit card shapes to programming languages.

When a company invents a new technology, it has an interest in its invention becoming a standard, so that other vendors produce tools that work with the invention and thus increase its likelihood of success. On the other hand, by handing over the invention to a standards committee, especially one that insists on a fair process, the company may lose control over the standard. For that reason, Sun Microsystems, the inventor of Java, never agreed to have a third-party organization standardize the Java language. They put in place their own standardization process, involving other companies but refusing to relinquish control. Another unfortunate but common tactic is to create a weak standard. For example, Netscape and Microsoft chose the European Computer Manufacturers Association (ECMA) to standardize the JavaScript language. ECMA was willing to settle for something less than truly useful, standardizing the behavior of the core language and just a few of its libraries.

Of course, many important pieces of technology aren't standardized at all. Consider the Windows operating system. Although Windows is often called a de-facto standard, it really is no standard at all. Nobody has ever attempted to define formally what the Windows operating system should do. The behavior changes at the whim of its vendor. That suits Microsoft just fine, because it makes it impossible for a third party to create its own version of Windows.

As a computer professional, there will be many times in your career when you need to make a decision whether to support a particular standard. Consider a simple example. In this chapter, you learn about the collection classes from the standard Java library. However, many computer scientists dislike these classes because of their numerous design issues. Should you use the Java collections in your own code, or should you implement a better set of collections? If you do the former, you have to deal with a design that is less than optimal. If you do the latter, other programmers may have a hard time understanding your code because they aren't familiar with your classes.

15.3 Sets

As you learned in Section 15.1, a **set** organizes its values in an order that is optimized for efficiency, which may not be the order in which you add elements. Inserting and removing elements is faster with a set than with a list.

In the following sections, you will learn how to choose a set implementation and how to work with sets.

15.3.1 Choosing a Set Implementation

The HashSet and TreeSet classes both implement the Set interface.

The `Set` interface in the standard Java library has the same methods as the `Collection` interface, shown in Table 1. However, there is an essential difference between arbitrary collections and sets. A set does not admit duplicates. If you add an element to a set that is already present, the insertion is ignored.

The `HashSet` and `TreeSet` classes implement the `Set` interface. These two classes provide set implementations based on two different mechanisms, called **hash tables** and **binary search trees**. Both implementations arrange the set elements so that finding, adding, and removing elements is fast, but they use different strategies.

Set implementations arrange the elements so that they can locate them quickly.

The basic idea of a hash table is simple. Set elements are grouped into smaller collections of elements that share the same characteristic. You can imagine a hash set of books as having a group for each color, so that books of the same color are in the same group. To find whether a book is already present, you just need to check it against the books in the same color group. Actually, hash tables don't use colors, but integer values (called hash codes) that can be computed from the elements.

In order to use a hash table, the elements must have a method to compute those integer values. This method is called `hashCode`. The elements must also belong to a class with a properly defined `equals` method (see Special Topic 9.7).

You can form hash sets holding objects of type `String`, `Integer`, `Double`, `Point`, `Rectangle`, or `Color`.

Many classes in the standard library implement these methods, for example `String`, `Integer`, `Double`, `Point`, `Rectangle`, `Color`, and all the collection classes. Therefore, you can form a `HashSet<String>`, `HashSet<Rectangle>`, or even a `HashSet<HashSet<Integer>>`.

Suppose you want to form a set of elements belonging to a class that you declared, such as a `HashSet<Book>`. Then you need to provide `hashCode` and `equals` methods for the class `Book`. There is one exception to this rule. If all elements are distinct (for example, if your program never has two `Book` objects with the same author and title), then you can simply inherit the `hashCode` and `equals` methods of the `Object` class.

On this shelf, books of the same color are grouped together. Similarly, in a hash table, objects with the same hash code are placed in the same group.

A tree set keeps its elements in sorted order.

The `TreeSet` class uses a different strategy for arranging its elements. Elements are kept in sorted order. For example, a set of books might be arranged by height, or alphabetically by author and title. The elements are not stored in an array—that would make adding and removing elements too slow. Instead, they are stored in nodes, as in a linked list. However, the nodes are not arranged in a linear sequence but in a tree shape.

You can form tree sets for any class that implements the `Comparable` interface, such as `String` or `Integer`.

In order to use a `TreeSet`, it must be possible to compare the elements and determine which one is "larger". You can use a `TreeSet` for classes such as `String` and `Integer` that implement the `Comparable` interface, which we discussed in Section 9.6.3. (That section also shows you how you can implement comparison methods for your own classes.)

As a rule of thumb, you should choose a `TreeSet` if you want to visit the set's elements in sorted order. Otherwise choose a `HashSet`—as long as the hash function is well chosen, it is a bit more efficient.

When you construct a `HashSet` or `TreeSet`, store the reference in a `Set<String>` variable, either as

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

or

```
Set<String> names = new TreeSet<String>();
```

After you construct the collection object, the implementation no longer matters; only the interface is important.

15.3.2 Working with Sets

Adding and removing set elements are accomplished with the `add` and `remove` methods:

```
names.add("Romeo");
names.remove("Juliet");
```

Sets don't have duplicates. Adding a duplicate of an element that is already present is ignored.

As in mathematics, a set collection in Java rejects duplicates. Adding an element has no effect if the element is already in the set. Similarly, attempting to remove an element that isn't in the set is ignored.

The `contains` method tests whether an element is contained in the set:

```
if (names.contains("Juliet")) . . .
```

Finally, to list all elements in the set, get an iterator. As with list iterators, you use the `next` and `hasNext` methods to step through the set.

```
Iterator<String> iter = names.iterator();
while (iter.hasNext())
{
   String name = iter.next();
   Do something with name
}
```

You can also use the "for each" loop instead of explicitly using an iterator:

```
for (String name : names)
{
   Do something with name
}
```

A set iterator visits the elements in the order in which the set implementation keeps them.

A set iterator visits the elements in the order in which the set implementation keeps them. This is not necessarily the order in which you inserted them. The order of elements in a hash set seems quite random because the hash code spreads the elements into different groups. When you visit elements of a tree set, they always appear in sorted order, even if you inserted them in a different order.

There is an important difference between the `Iterator` that you obtain from a set and the `ListIterator` that a list yields. The `ListIterator` has an `add` method to add an element at the list iterator position. The `Iterator` interface has no such method. It makes no sense to add an element at a particular position in a set, because the set can order the elements any way it likes. Thus, you always add elements directly to a set, never to an iterator of the set.

You cannot add an element to a set at an iterator position.

However, you can remove a set element at an iterator position, just as you do with list iterators.

Also, the `Iterator` interface has no `previous` method to go backward through the elements. Because the elements are not ordered, it is not meaningful to distinguish between "going forward" and "going backward".

Table 4 Working with Sets

`Set<String> names;`	Use the interface type for variable declarations.
`names = new HashSet<String>();`	Use a `TreeSet` if you need to visit the elements in sorted order.
`names.add("Romeo");`	Now `names.size()` is 1.
`names.add("Fred");`	Now `names.size()` is 2.
`names.add("Romeo");`	`names.size()` is still 2. You can't add duplicates.
`if (names.contains("Fred"))`	The `contains` method checks whether a value is contained in the set. In this case, the method returns `true`.
`System.out.println(names);`	Prints the set in the format `[Fred, Romeo]`. The elements need not be shown in the order in which they were inserted.

`for (String name : names)`
`{`
`  . . .`
`}`
	Use this loop to visit all elements of a set.

`names.remove("Romeo");`	Now `names.size()` is 1.
`names.remove("Juliet");`	It is not an error to remove an element that is not present. The method call has no effect.

The following program shows a practical application of sets. It reads in all words from a dictionary file that contains correctly spelled words and places them in a set. It then reads all words from a document—here, the book *Alice in Wonderland*—into a second set. Finally, it prints all words from that set that are not in the dictionary set. These are the potential misspellings. (As you can see from the output, we used an American dictionary, and words with British spelling, such as *clamour*, are flagged as potential errors.)

section_3/SpellCheck.java

```
1  import java.util.HashSet;
2  import java.util.Scanner;
3  import java.util.Set;
4  import java.io.File;
5  import java.io.FileNotFoundException;
6
7  /**
8     This program checks which words in a file are not present in a dictionary.
9  */
10 public class SpellCheck
11 {
12    public static void main(String[] args)
13       throws FileNotFoundException
14    {
15       // Read the dictionary and the document
16
17       Set<String> dictionaryWords = readWords("words");
18       Set<String> documentWords = readWords("alice30.txt");
19
20       // Print all words that are in the document but not the dictionary
21
22       for (String word : documentWords)
23       {
24          if (!dictionaryWords.contains(word))
25          {
26             System.out.println(word);
27          }
28       }
29    }
30
31    /**
32       Reads all words from a file.
33       @param filename the name of the file
34       @return a set with all lowercased words in the file. Here, a
35       word is a sequence of upper- and lowercase letters.
36    */
37    public static Set<String> readWords(String filename)
38       throws FileNotFoundException
39    {
40       Set<String> words = new HashSet<String>();
41       Scanner in = new Scanner(new File(filename));
42       // Use any characters other than a-z or A-Z as delimiters
43       in.useDelimiter("[^a-zA-Z]+");
44       while (in.hasNext())
45       {
46          words.add(in.next().toLowerCase());
47       }
```

```
48          return words;
49       }
50    }
```

Program Run

```
neighbouring
croqueted
pennyworth
dutchess
comfits
xii
dinn
clamour
...
```

10. Arrays and lists remember the order in which you added elements; sets do not. Why would you want to use a set instead of an array or list?
11. Why are set iterators different from list iterators?
12. What is wrong with the following test to check whether the `Set<String> s` contains the elements `"Tom"`, `"Diana"`, and `"Harry"`?

    ```
    if (s.toString().equals("[Tom, Diana, Harry]")) . . .
    ```

13. How can you correctly implement the test of Self Check 12?
14. Write a loop that prints all elements that are in both `Set<String> s` and `Set<String> t`.
15. Suppose you changed line 40 of the `SpellCheck` program to use a `TreeSet` instead of a `HashSet`. How would the output change?

Practice It Now you can try these exercises at the end of the chapter: P15.7, P15.8, P15.13.

Programming Tip 15.1

Use Interface References to Manipulate Data Structures

It is considered good style to store a reference to a `HashSet` or `TreeSet` in a variable of type `Set`:

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

This way, you have to change only one line if you decide to use a `TreeSet` instead.

If a method can operate on arbitrary collections, use the `Collection` interface type for the parameter variable:

```
public static void removeLongWords(Collection<String> words)
```

In theory, we should make the same recommendation for the `List` interface, namely to save `ArrayList` and `LinkedList` references in variables of type `List`. However, the `List` interface has `get` and `set` methods for random access, even though these methods are very inefficient for linked lists. You can't write efficient code if you don't know whether the methods that you are calling are efficient or not. This is plainly a serious design error in the standard library, and it makes the `List` interface somewhat unattractive.

15.4 Maps

The HashMap and TreeMap classes both implement the Map interface.

A **map** allows you to associate elements from a *key set* with elements from a *value collection*. You use a map when you want to look up objects by using a key. For example, Figure 10 shows a map from the names of people to their favorite colors.

Just as there are two kinds of set implementations, the Java library has two implementations for the Map interface: HashMap and TreeMap.

After constructing a HashMap or TreeMap, you can store the reference to the map object in a Map reference:

```
Map<String, Color> favoriteColors = new HashMap<String, Color>();
```

Use the put method to add an association:

```
favoriteColors.put("Juliet", Color.RED);
```

You can change the value of an existing association, simply by calling put again:

```
favoriteColors.put("Juliet", Color.BLUE);
```

The get method returns the value associated with a key.

```
Color julietsFavoriteColor = favoriteColors.get("Juliet");
```

If you ask for a key that isn't associated with any values, then the get method returns null.

To remove an association, call the remove method with the key:

```
favoriteColors.remove("Juliet");
```

Table 5 Working with Maps

`Map<String, Integer> scores;`	Keys are strings, values are Integer wrappers. Use the interface type for variable declarations.
`scores = new TreeMap<String, Integer>();`	Use a HashMap if you don't need to visit the keys in sorted order.
`scores.put("Harry", 90);` `scores.put("Sally", 95);`	Adds keys and values to the map.
`scores.put("Sally", 100);`	Modifies the value of an existing key.
`int n = scores.get("Sally");` `Integer n2 = scores.get("Diana");`	Gets the value associated with a key, or null if the key is not present. n is 100, n2 is null.
`System.out.println(scores);`	Prints scores.toString(), a string of the form {Harry=90, Sally=100}

`for (String key : scores.keySet())`
`{`
`  Integer value = scores.get(key);`
`  . . .`
`}`
	Iterates through all map keys and values.

`scores.remove("Sally");`	Removes the key and value.

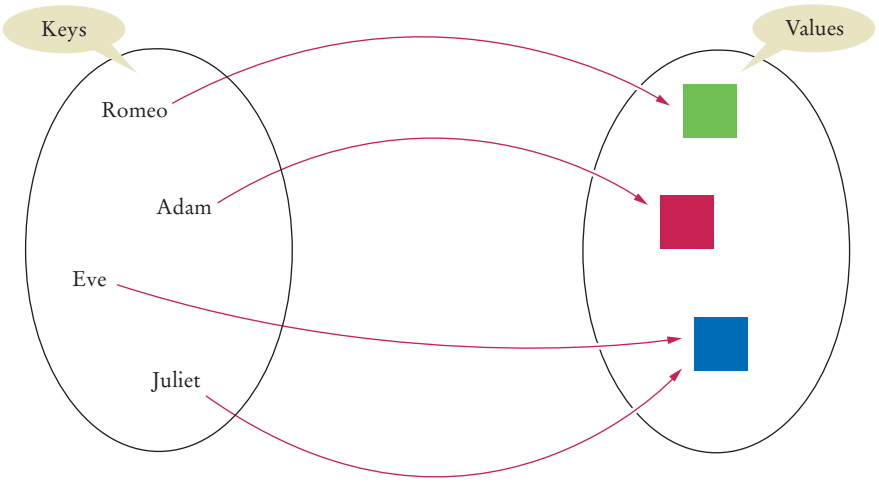


Figure 10 A Map

To find all keys and values in a map, iterate through the key set and find the values that correspond to the keys.

Sometimes you want to enumerate all keys in a map. The `keySet` method yields the set of keys. You can then ask the key set for an iterator and get all keys. From each key, you can find the associated value with the `get` method. Thus, the following instructions print all key/value pairs in a map `m`:

```
Set<String> keySet = m.keySet();
for (String key : keySet)
{
   Color value = m.get(key);
   System.out.println(key + "->" + value);
}
```

This sample program shows a map in action:

section_4/MapDemo.java

```
1  import java.awt.Color;
2  import java.util.HashMap;
3  import java.util.Map;
4  import java.util.Set;
5
6  /**
7     This program demonstrates a map that maps names to colors.
8  */
9  public class MapDemo
10 {
11    public static void main(String[] args)
12    {
13       Map<String, Color> favoriteColors = new HashMap<String, Color>();
14       favoriteColors.put("Juliet", Color.BLUE);
15       favoriteColors.put("Romeo", Color.GREEN);
16       favoriteColors.put("Adam", Color.RED);
17       favoriteColors.put("Eve", Color.BLUE);
18
19       // Print all keys and values in the map
20
21       Set<String> keySet = favoriteColors.keySet();
```

```
22         for (String key : keySet)
23         {
24            Color value = favoriteColors.get(key);
25            System.out.println(key + " : " + value);
26         }
27      }
28   }
```

Program Run

```
Juliet : java.awt.Color[r=0,g=0,b=255]
Adam : java.awt.Color[r=255,g=0,b=0]
Eve : java.awt.Color[r=0,g=0,b=255]
Romeo : java.awt.Color[r=0,g=255,b=0]
```

16. What is the difference between a set and a map?
17. Why is the collection of the keys of a map a set and not a list?
18. Why is the collection of the values of a map not a set?
19. Suppose you want to track how many times each word occurs in a document. Declare a suitable map variable.
20. What is a `Map<String, HashSet<String>>`? Give a possible use for such a structure.

Practice It Now you can try these exercises at the end of the chapter: R15.17, P15.9, P15.14.

HOW TO 15.1

Choosing a Collection

Suppose you need to store objects in a collection. You have now seen a number of different data structures. This How To reviews how to pick an appropriate collection for your application.

Step 1 Determine how you access the values.

You store values in a collection so that you can later retrieve them. How do you want to access individual values? You have several choices:

- Values are accessed by an integer position. Use an `ArrayList`.
- Values are accessed by a key that is not a part of the object. Use a map.
- Values are accessed only at one of the ends. Use a queue (for first-in, first-out access) or a stack (for last-in, first-out access).
- You don't need to access individual values by position. Refine your choice in Steps 3 and 4.

Step 2 Determine the element types or key/value types.

For a list or set, determine the type of the elements that you want to store. For example, if you collect a set of books, then the element type is `Book`.

Similarly, for a map, determine the types of the keys and the associated values. If you want to look up books by ID, you can use a `Map<Integer, Book>` or `Map<String, Book>`, depending on your ID type.

Step 3 Determine whether element or key order matters.

When you visit elements from a collection or keys from a map, do you care about the order in which they are visited? You have several choices:

- Elements or keys must be sorted. Use a `TreeSet` or `TreeMap`. Go to Step 6.
- Elements must be in the same order in which they were inserted. Your choice is now narrowed down to a `LinkedList` or an `ArrayList`.
- It doesn't matter. As long as you get to visit all elements, you don't care in which order. If you chose a map in Step 1, use a `HashMap` and go to Step 5.

Step 4 For a collection, determine which operations must be fast.

You have several choices:

- Finding elements must be fast. Use a `HashSet`.
- It must be fast to add or remove elements at the beginning, or, provided that you are already inspecting an element there, another position. Use a `LinkedList`.
- You only insert or remove at the end, or you collect so few elements that you aren't concerned about speed. Use an `ArrayList`.

Step 5 For hash sets and maps, decide whether you need to implement the `hashCode` and `equals` methods.

- If your elements or keys belong to a class that someone else implemented, check whether the class has its own `hashCode` and `equals` methods. If so, you are all set. This is the case for most classes in the standard Java library, such as `String`, `Integer`, `Rectangle`, and so on.
- If not, decide whether you can compare the elements by identity. This is the case if you never construct two distinct elements with the same contents. In that case, you need not do anything—the `hashCode` and `equals` methods of the `Object` class are appropriate.
- Otherwise, you need to implement your own `equals` and `hashCode` methods—see Special Topics 9.7 and Special Topic 15.1.

Step 6 If you use a tree, decide whether to supply a comparator.

Look at the class of the set elements or map keys. Does that class implement the `Comparable` interface? If so, is the sort order given by the `compareTo` method the one you want? If yes, then you don't need to do anything further. This is the case for many classes in the standard library, in particular for `String` and `Integer`.

If not, then your element class must implement the `Comparable` interface (Section 9.6.3), or you must declare a class that implements the `Comparator` interface (see Special Topic 14.5).

WORKED EXAMPLE 15.1 Word Frequency

In this Worked Example, we read a text file and print a list of all words in the file in alphabetical order, together with a count that indicates how often each word occurred in the file.

⊕ Available online in WileyPLUS and at `www.wiley.com/college/horstmann`.

Special Topic 15.1

Hash Functions

If you use a hash set or hash map with your own classes, you may need to implement a hash function. A **hash function** is a function that computes an integer value, the **hash code**, from an object in such a way that different objects are likely to yield different hash codes. Because hashing is so important, the `Object` class has a `hashCode` method. The call

```
int h = x.hashCode();
```

computes the hash code of any object `x`. If you want to put objects of a given class into a `HashSet` or use the objects as keys in a `HashMap`, the class should override this method. The method should be implemented so that different objects are likely to have different hash codes.

A good hash function produces different hash values for each object so that they are scattered about in a hash table.

A hash function computes an integer value from an object.

For example, the `String` class declares a hash function for strings that does a good job of producing different integer values for different strings. Table 6 shows some examples of strings and their hash codes.

A good hash function minimizes *collisions*—identical hash codes for different objects.

It is possible for two or more distinct objects to have the same hash code; this is called a *collision*. For example, the strings `"Ugh"` and `"VII"` happen to have the same hash code, but these collisions are very rare for strings (see Exercise P15.15).

The `hashCode` method of the `String` class combines the characters of a string into a numerical code. The code isn't simply the sum of the character values—that would not scramble the character values enough. Strings that are permutations of another (such as `"eat"` and `"tea"`) would all have the same hash code.

Here is the method the standard library uses to compute the hash code for a string:

```
final int HASH_MULTIPLIER = 31;
int h = 0;
for (int i = 0; i < s.length(); i++)
{
   h = HASH_MULTIPLIER * h + s.charAt(i);
}
```

For example, the hash code of `"eat"` is

```
31 * (31 * 'e' + 'a') + 't' = 100184
```

Table 6 Sample Strings and Their Hash Codes

String	Hash Code
`"eat"`	100184
`"tea"`	114704
`"Juliet"`	–2065036585
`"Ugh"`	84982
`"VII"`	84982

The hash code of "tea" is quite different, namely

```
31 * (31 * 't' + 'e') + 'a' = 114704
```

(Use the Unicode table from Appendix A to look up the character values: 'a' is 97, 'e' is 101, and 't' is 116.)

Override hashCode methods in your own classes by combining the hash codes for the instance variables.

For your own classes, you should make up a hash code that combines the hash codes of the instance variables in a similar way. For example, let us declare a hashCode method for the Country class from Section 9.6.

There are two instance variables: the country name and the area. First, compute their hash codes. You know how to compute the hash code of a string. To compute the hash code of a floating-point number, first wrap the floating-point number into a Double object, and then compute its hash code.

```
public class Country
{
   public int hashCode()
   {
      int h1 = name.hashCode();
      int h2 = new Double(area).hashCode();
      . . .
   }
}
```

Then combine the two hash codes:

```
final int HASH_MULTIPLIER = 29;
int h = HASH_MULTIPLIER * h1 + h2;
return h;
```

Use a prime number as the hash multiplier—it scrambles the values well.

If you have more than two instance variables, then combine their hash codes as follows:

```
int h = HASH_MULTIPLIER * h1 + h2;
h = HASH_MULTIPLIER * h + h3;
h = HASH_MULTIPLIER * h + h4;
. . .
return h;
```

If one of the instance variables is an integer, just use the value as its hash code.

When you supply your own hashCode method for a class, you must also provide a compatible equals method. The equals method is used to differentiate between two objects that happen to have the same hash code.

A class's hashCode method must be compatible with its equals method.

The equals and hashCode methods must be *compatible* with each other. Two objects that are equal must yield the same hash code.

ONLINE EXAMPLE

A program that demonstrates a hash set with objects of the Country class.

You get into trouble if your class declares an equals method but not a hashCode method. Suppose the Country class declares an equals method (checking that the name and area are the same), but no hashCode method. Then the hashCode method is inherited from the Object superclass. That method computes a hash code from the *memory location* of the object. Then it is very likely that two objects with the same contents will have different hash codes, in which case a hash set will store them as two distinct objects.

However, if you declare *neither* equals *nor* hashCode, then there is no problem. The equals method of the Object class considers two objects equal only if their memory location is the same. That is, the Object class has compatible equals and hashCode methods. Of course, then the notion of equality is very restricted: Only identical objects are considered equal. That can be a perfectly valid notion of equality, depending on your application.

15.5 Stacks, Queues, and Priority Queues

In the following sections, we cover stacks, queues, and priority queues. These data structures each have a different policy for data removal. Removing an element yields the most recently added element, the least recently added, or the element with the highest priority.

15.5.1 Stacks

A stack is a collection of elements with "last-in, first-out" retrieval.

A **stack** lets you insert and remove elements only at one end, traditionally called the *top* of the stack. New items can be added to the top of the stack. Items are removed from the top of the stack as well. Therefore, they are removed in the order that is opposite from the order in which they have been added, called *last-in, first-out* or *LIFO* order. For example, if you add items `A`, `B`, and `C` and then remove them, you obtain `C`, `B`, and `A`. With stacks, the addition and removal operations are called `push` and `pop`.

The last pancake that has been added to this stack will be the first one that is consumed.

```
Stack<String> s = new Stack<String>();
s.push("A"); s.push("B"); s.push("C");
while (s.size() > 0)
{
   System.out.print(s.pop() + " "); // Prints C B A
}
```

The Undo key pops commands off a stack, so that the last command is the first to be undone.

There are many applications for stacks in computer science. Consider the undo feature of a word processor. It keeps the issued commands in a stack. When you select "Undo", the *last* command is undone, then the next-to-last, and so on.

Another important example is the **run-time stack** that a processor or virtual machine keeps to store the values of variables in nested methods. Whenever a new method is called, its parameter variables and local variables are pushed onto a stack. When the method exits, they are popped off again.

You will see other applications in Section 15.6.

The Java library provides a simple `Stack` class with methods `push`, `pop`, and `peek`—the latter gets the top element of the stack but does not remove it (see Table 7).

Table 7 Working with Stacks

`Stack<Integer> s = new Stack<Integer>();`	Constructs an empty stack.

`s.push(1);`
`s.push(2);`
`s.push(3);`
	Adds to the top of the stack; `s` is now `[1, 2, 3]`. (Following the `toString` method of the `Stack` class, we show the top of the stack at the end.)

`int top = s.pop();`	Removes the top of the stack; `top` is set to 3 and `s` is now `[1, 2]`.
`head = s.peek();`	Gets the top of the stack without removing it; `head` is set to 2.

15.5.2 Queues

A queue is a collection of elements with "first-in, first-out" retrieval.

To visualize a queue, think of people lining up.

A **queue** lets you add items to one end of the queue (the *tail*) and remove them from the other end of the queue (the *head*). Queues yield items in a *first-in, first-out* or *FIFO* fashion. Items are removed in the same order in which they were added.

A typical application is a print queue. A printer may be accessed by several applications, perhaps running on different computers. If each of the applications tried to access the printer at the same time, the printout would be garbled. Instead, each application places its print data into a file and adds that file to the print queue. When the printer is done printing one file, it retrieves the next one from the queue. Therefore, print jobs are printed using the "first-in, first-out" rule, which is a fair arrangement for users of the shared printer.

The `Queue` interface in the standard Java library has methods `add` to add an element to the tail of the queue, `remove` to remove the head of the queue, and `peek` to get the head element of the queue without removing it (see Table 8).

The `LinkedList` class implements the `Queue` interface. Whenever you need a queue, simply initialize a `Queue` variable with a `LinkedList` object:

```
Queue<String> q = new LinkedList<String>();
q.add("A"); q.add("B"); q.add("C");
while (q.size() > 0) { System.out.print(q.remove() + " "); } // Prints A B C
```

The standard library provides several queue classes that we do not discuss in this book. Those classes are intended for work sharing when multiple activities (called threads) run in parallel.

Table 8 Working with Queues

`Queue<Integer> q = new LinkedList<Integer>();`	The `LinkedList` class implements the `Queue` interface.

`q.add(1);`
`q.add(2);`
`q.add(3);`
	Adds to the tail of the queue; `q` is now `[1, 2, 3]`.

`int head = q.remove();`	Removes the head of the queue; `head` is set to 1 and `q` is `[2, 3]`.
`head = q.peek();`	Gets the head of the queue without removing it; `head` is set to 2.

15.5.3 Priority Queues

When removing an element from a priority queue, the element with the most urgent priority is retrieved.

A **priority queue** collects elements, each of which has a *priority*. A typical example of a priority queue is a collection of work requests, some of which may be more urgent than others. Unlike a regular queue, the priority queue does not maintain a first-in, first-out discipline. Instead, elements are retrieved according to their priority. In other words, new items can be inserted in any order. But whenever an item is removed, it is the item with the most urgent priority.

When you retrieve an item from a priority queue, you always get the most urgent one.

It is customary to give low values to urgent priorities, with priority 1 denoting the most urgent priority. Thus, each removal operation extracts the *minimum* element from the queue.

For example, consider this code in which we add objects of a class `WorkOrder` into a priority queue. Each work order has a priority and a description.

```
PriorityQueue<WorkOrder> q = new PriorityQueue<WorkOrder>();
q.add(new WorkOrder(3, "Shampoo carpets"));
q.add(new WorkOrder(1, "Fix broken sink"));
q.add(new WorkOrder(2, "Order cleaning supplies"));
```

When calling `q.remove()` for the first time, the work order with priority 1 is removed. The next call to `q.remove()` removes the work order whose priority is highest among those remaining in the queue—in our example, the work order with priority 2. If there happen to be two elements with the same priority, the priority queue will break ties arbitrarily.

Because the priority queue needs to be able to tell which element is the smallest, the added elements should belong to a class that implements the `Comparable` interface. (See Section 9.6.3 for a description of that interface type.)

Table 9 shows the methods of the `PriorityQueue` class in the standard Java library.

ONLINE EXAMPLE

Programs that demonstrate stacks, queues, and priority queues.

Table 9 Working with Priority Queues

`PriorityQueue<Integer> q = new PriorityQueue<Integer>();`	This priority queue holds `Integer` objects. In practice, you would use objects that describe tasks.
`q.add(3); q.add(1); q.add(2);`	Adds values to the priority queue.
`int first = q.remove(); int second = q.remove();`	Each call to `remove` removes the lowest priority item: `first` is set to 1, `second` to 2.
`int next = q.peek();`	Gets the smallest value in the priority queue without removing it.

SELF CHECK

21. Why would you want to declare a variable as

```
Queue<String> q = new LinkedList<String>()
```

instead of simply declaring it as a linked list?

22. Why wouldn't you want to use an array list for implementing a queue?

23. What does this code print?

```
Queue<String> q = new LinkedList<String>();
q.add("A");
q.add("B");
q.add("C");
while (q.size() > 0) { System.out.print(q.remove() + " "); }
```

24. Why wouldn't you want to use a stack to manage print jobs?

25. In the sample code for a priority queue, we used a `WorkOrder` class. Could we have used strings instead?

```
PriorityQueue<String> q = new PriorityQueue<String>();
q.add("3 - Shampoo carpets");
q.add("1 - Fix broken sink");
q.add("2 - Order cleaning supplies");
```

Practice It Now you can try these exercises at the end of the chapter: R15.12, P15.3, P15.4.

15.6 Stack and Queue Applications

Stacks and queues are, despite their simplicity, very versatile data structures. In the following sections, you will see some of their most useful applications.

15.6.1 Balancing Parentheses

A stack can be used to check whether parentheses in an expression are balanced.

In Common Error 2.5, you saw a simple trick for detecting unbalanced parentheses in an expression such as

```
-(b * b - (4 * a * c ) ) / (2 * a)
 1       2          1 0   1     0
```

Increment a counter when you see a `(` and decrement it when you see a `)`. The counter should never be negative, and it should be zero at the end of the expression.

That works for expressions in Java, but in mathematical notation, one can have more than one kind of parentheses, such as

$$-\{ [b \cdot b - (4 \cdot a \cdot c)] / (2 \cdot a) \}$$

To see whether such an expression is correctly formed, place the parentheses on a stack:

```
When you see an opening parenthesis, push it on the stack.
When you see a closing parenthesis, pop the stack.
If the opening and closing parentheses don't match
   The parentheses are unbalanced. Exit.
If at the end the stack is empty
   The parentheses are balanced.
Else
   The parentheses are not balanced.
```

ONLINE EXAMPLE

A program for checking balanced parentheses.

Here is a walkthrough of the sample expression:

Stack	Unread expression	Comments
Empty	-{ [b * b - (4 * a * c)] / (2 * a) }	
{	[b * b - (4 * a * c)] / (2 * a) }	
{ [	b * b - (4 * a * c)] / (2 * a) }	
{ [(	4 * a * c)] / (2 * a) }	
{ [	] / (2 * a) }	(matches)
{	/ (2 * a) }	[matches]
{ (	2 * a) }	
{	}	(matches)
Empty	No more input	{ matches }
		The parentheses are balanced

15.6.2 Evaluating Reverse Polish Expressions

Use a stack to evaluate expressions in reverse Polish notation.

Consider how you write arithmetic expressions, such as (3 + 4) × 5. The parentheses are needed so that 3 and 4 are added before multiplying the result by 5.

However, you can eliminate the parentheses if you write the operators *after* the numbers, like this: 3 4 + 5 × (see Random Fact 15.2 on page W701). To evaluate this expression, apply + to 3 and 4, yielding 7, and then simplify 7 5 × to 35. It gets trickier for complex expressions. For example, 3 4 5 + × means to compute 4 5 + (that is, 9), and then evaluate 3 9 ×. If we evaluate this expression left-to-right, we need to leave the 3 somewhere while we work on 4 5 +. Where? We put it on a stack. The algorithm for evaluating reverse Polish expressions is simple:

```
If you read a number
   Push it on the stack.
Else if you read an operand
   Pop two values off the stack.
   Combine the values with the operand.
   Push the result back onto the stack.
Else if there is no more input
   Pop and display the result.
```

Here is a walkthrough of evaluating the expression 3 4 5 + ×:

Stack	Unread expression	Comments
Empty	3 4 5 + x	
3	4 5 + x	Numbers are pushed on the stack
3 4	5 + x	
3 4 5	+ x	
3 9	x	Pop 4 and 5, push 4 5 +
27	No more input	Pop 3 and 9, push 3 9 x
Empty		Pop and display the result, 27

The following program simulates a reverse Polish calculator:

section_6_2/Calculator.java

```
1  import java.util.Scanner;
2  import java.util.Stack;
3
4  /**
5     This calculator uses the reverse Polish notation.
6  */
7  public class Calculator
8  {
9     public static void main(String[] args)
10    {
11       Scanner in = new Scanner(System.in);
12       Stack<Integer> results = new Stack<Integer>();
13       System.out.println("Enter one number or operator per line, Q to quit. ");
14       boolean done = false;
```

```
15          while (!done)
16          {
17             String input = in.nextLine();
18
19             // If the command is an operator, pop the arguments and push the result
20
21             if (input.equals("+"))
22             {
23                results.push(results.pop() + results.pop());
24             }
25             else if (input.equals("-"))
26             {
27                Integer arg2 = results.pop();
28                results.push(results.pop() - arg2);
29             }
30             else if (input.equals("*") || input.equals("x"))
31             {
32                results.push(results.pop() * results.pop());
33             }
34             else if (input.equals("/"))
35             {
36                Integer arg2 = results.pop();
37                results.push(results.pop() / arg2);
38             }
39             else if (input.equals("Q") || input.equals("q"))
40             {
41                done = true;
42             }
43             else
44             {
45                // Not an operator--push the input value
46
47                results.push(Integer.parseInt(input));
48             }
49             System.out.println(results);
50          }
51       }
52    }
```

15.6.3 Evaluating Algebraic Expressions

Using two stacks, you can evaluate expressions in standard algebraic notation.

In the preceding section, you saw how to evaluate expressions in reverse Polish notation, using a single stack. If you haven't found that notation attractive, you will be glad to know that one can evaluate an expression in the standard algebraic notation using two stacks—one for numbers and one for operators.

Use two stacks to evaluate algebraic expressions.

First, consider a simple example, the expression 3 + 4. We push the numbers on the number stack and the operators on the operator stack. Then we pop both numbers and the operator, combine the numbers with the operator, and push the result.

	Number stack Empty	Operator stack Empty	Unprocessed input 3 + 4	Comments
1	3		+ 4	
2	3	+	4	
3	4 3	+	No more input	Evaluate the top.
4	7			The result is 7.

This operation is fundamental to the algorithm. We call it "evaluating the top".

In algebraic notation, each operator has a *precedence*. The + and - operators have the lowest precedence, * and / have a higher (and equal) precedence.

Consider the expression 3 × 4 + 5. Here are the first processing steps:

	Number stack Empty	Operator stack Empty	Unprocessed input 3 × 4 + 5	Comments
1	3		× 4 + 5	
2	3	×	4 + 5	
3	4 3	×	+ 5	Evaluate × before +.

Because × has a higher precedence than +, we are ready to evaluate the top:

	Number stack	Operator stack		Comments
4	12	+	5	
5	5 12	+	No more input	Evaluate the top.
6	17			That is the result.

With the expression, 3 + 4 × 5, we add × to the operator stack because we must first read the next number; then we can evaluate × and then the +:

	Number stack Empty	Operator stack Empty	Unprocessed input 3 + 4 × 5	Comments
1	3		+ 4 × 5	
2	3	+	4 + 5	

3	4 3	+	× 5	Don't evaluate + yet.
4	4 3	× +	5	

In other words, we keep operators on the stack until they are ready to be evaluated. Here is the remainder of the computation:

	Number stack	Operator stack		Comments
5	5 4 3	× +	No more input	Evaluate the top.
6	20 3	+		Evaluate top again.
7	23			That is the result.

To see how parentheses are handled, consider the expression 3 × (4 + 5). A (is pushed on the operator stack. The + is pushed as well. When we encounter the), we know that we are ready to evaluate the top until the matching (reappears:

	Number stack Empty	Operator stack Empty	Unprocessed input 3 × (4 + 5)	Comments
1	3		× (4 + 5)	
2	3	×	(4 + 5)	
3	3	(×	4 + 5)	Don't evaluate × yet.
4	4 3	(×	+ 5)	
5	4 3	+ (×	5)	
6	5 4 3	+ (×	)	Evaluate the top.
7	9 3	(×	No more input	Pop (.
8	9 3	×		Evaluate top again.
9	27			That is the result.

Here is the algorithm:

```
If you read a number
   Push it on the number stack.
Else if you read a (
   Push it on the operator stack.
Else if you read an operator op
   While the top of the stack has a higher precedence than op
      Evaluate the top.
   Push op on the operator stack.
Else if you read a )
   While the top of the stack is not a (
      Evaluate the top.
   Pop the (.
Else if there is no more input
   While the operator stack is not empty
      Evaluate the top.
```

At the end, the remaining value on the number stack is the value of the expression.

The algorithm makes use of this helper method that evaluates the topmost operator with the topmost numbers:

```
Evaluate the top:
Pop two numbers off the number stack.
Pop an operator off the operator stack.
Combine the numbers with that operator.
Push the result on the number stack.
```

ONLINE EXAMPLE

The complete code for the expression calculator.

15.6.4 Backtracking

Use a stack to remember choices you haven't yet made so that you can backtrack to them.

Suppose you are inside a maze. You need to find the exit. What should you do when you come to an intersection? You can continue exploring one of the paths, but you will want to remember the other ones. If your chosen path didn't work, you can go back to one of the other choices and try again.

A stack can be used to track positions in a maze.

Of course, as you go along one path, you may reach further intersections, and you need to remember your choice again. Simply use a stack to remember the paths that still need to be tried. The process of returning to a choice point and trying another choice is called *backtracking*. By using a stack, you return to your more recent choices before you explore the earlier ones.

Figure 11 shows an example. We start at a point in the maze, at position (3, 4). There are four possible paths. We push them all on a stack 1. We pop off the topmost one, traveling north from (3, 4). Following this path leads to position (1, 4). We now push two choices on the stack, going west or east 2. Both of them lead to dead ends 3 4.

Now we pop off the path from (3,4) going east. That too is a dead end 5. Next is the path from (3, 4) going south. At (5, 4), it comes to an intersection. Both choices are pushed on the stack 6. They both lead to dead ends 7 8.

Finally, the path from (3, 4) going west leads to an exit 9.

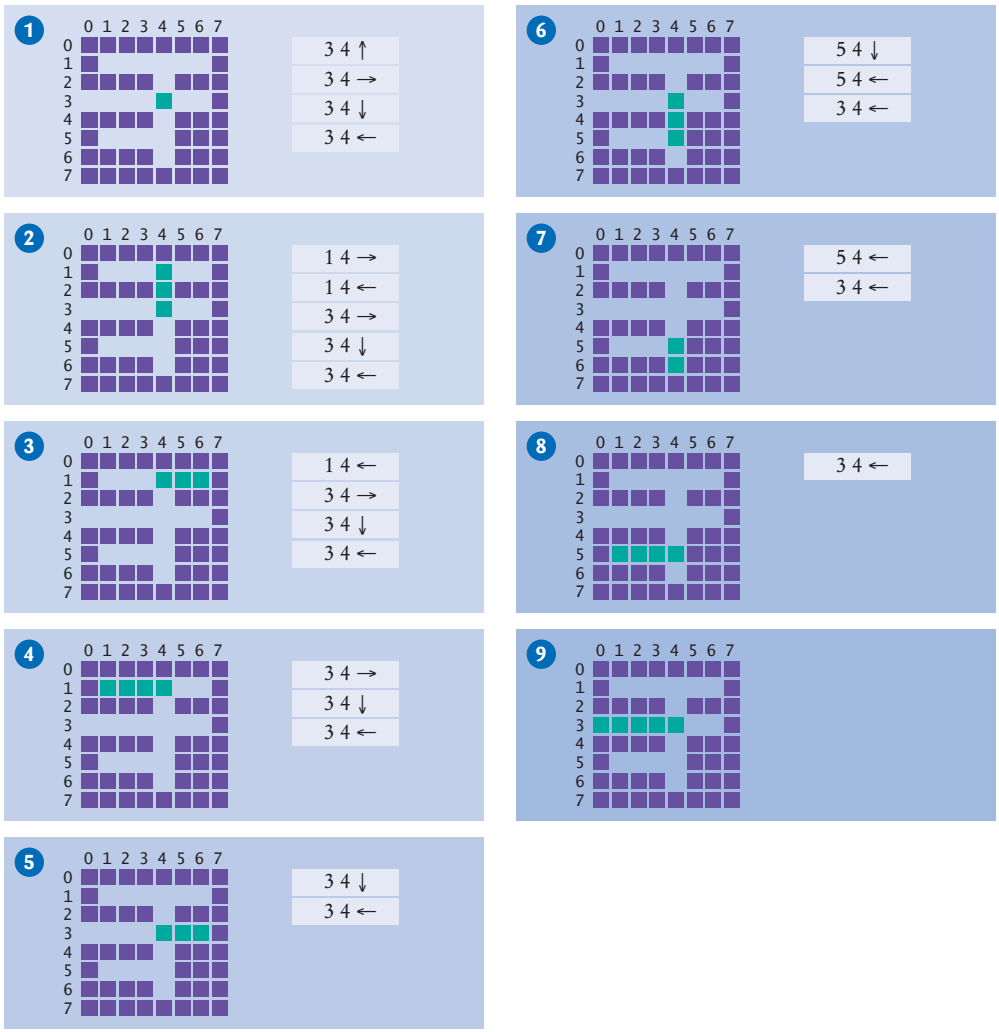


Figure 11 Backtracking Through a Maze

Using a stack, we have found a path out of the maze. Here is the pseudocode for our maze-finding algorithm:

```
Push all paths from the point on which you are standing on a stack.
While the stack is not empty
   Pop a path from the stack.
   Follow the path until you reach an exit, intersection, or dead end.
   If you found an exit
      Congratulations!
   Else if you found an intersection
      Push all paths meeting at the intersection, except the current one, onto the stack.
```

This algorithm will find an exit from the maze, provided that the maze has no *cycles*. If it is possible that you can make a circle and return to a previously visited intersection along a different sequence of paths, then you need to work harder—see Exercise P15.25.

How you implement this algorithm depends on the description of the maze. In the example code, we use a two-dimensional array of characters, with spaces for corridors and asterisks for walls, like this:

```
* * * * * * * *
*             *
* * * *   * * *
*               *
* * * *   * * *
*       * * *
* * * *   * * *
* * * * * * * *
```

In the example code, a `Path` object is constructed with a starting position and a direction (North, East, South, or West). The `Maze` class has a method that extends a path until it reaches an intersection or exit, or until it is blocked by a wall, and a method that computes all paths from an intersection point.

ONLINE EXAMPLE

A complete program demonstrating backtracking.

Note that you can use a queue instead of a stack in this algorithm. Then you explore the earlier alternatives before the later ones. This can work just as well for finding an answer, but it isn't very intuitive in the context of exploring a maze—you would have to imagine being teleported back to the initial intersections rather than just walking back to the last one.

26. What is the value of the reverse Polish notation expression 2 3 4 + 5 × ×?

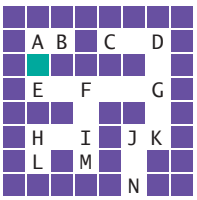
27. Why does the branch for the subtraction operator in the `Calculator` program not simply execute

```
results.push(results.pop() - results.pop());
```

28. In the evaluation of the expression 3 – 4 + 5 with the algorithm of Section 15.6.3, which operator gets evaluated first?

29. In the algorithm of Section 15.6.3, are the operators on the operator stack always in increasing precedence?

30. Consider the following simple maze. Assuming that we start at the marked point and push paths in the order West, South, East, North, in which order are the lettered points visited, using the algorithm of Section 15.6.4?



Practice It Now you can try these exercises at the end of the chapter: R15.21, P15.21, P15.22, P15.25, P15.26.

WORKED EXAMPLE 15.2 Simulating a Queue of Waiting Customers

This Worked Example shows how to use a queue to simulate an actual queue of waiting customers.

Available online in WileyPLUS and at `www.wiley.com/college/horstmann`.

Random Fact 15.2 Reverse Polish Notation

In the 1920s, the Polish mathematician Jan Łukasiewicz realized that it is possible to dispense with parentheses in arithmetic expressions, provided that you write the operators *before* their arguments, for example, + 3 4 instead of 3 + 4. Thirty years later, Australian computer scientist Charles Hamblin noted that an even better scheme would be to have the operators *follow* the operands. This was termed **reverse Polish notation** or RPN.

Standard Notation	Reverse Polish Notation
3 + 4	3 4 +
3 + 4 × 5	3 4 5 × +
3 × (4 + 5)	3 4 5 + ×
(3 + 4) × (5 + 6)	3 4 + 5 6 + ×
3 + 4 + 5	3 4 + 5 +

Reverse Polish notation might look strange to you, but that is just an accident of history. Had earlier mathematicians realized its advantages, today's schoolchildren might be using it and not worrying about precedence rules and parentheses.

In 1972, Hewlett-Packard introduced the HP 35 calculator that used reverse Polish notation. The calculator had no keys labeled with parentheses or an equals symbol. There is just a key labeled ENTER to push a number onto a stack. For that reason, Hewlett-Packard's marketing department used to refer to their product as "the calculators that have no equal".

Over time, calculator vendors have adapted to the standard algebraic notation rather than forcing its users to learn a new notation. However, those users who have made the effort to learn reverse Polish notation tend to be fanatic proponents, and to this day, some Hewlett-Packard calculator models still support it.

The Calculator with No Equal

VIDEO EXAMPLE 15.1 Building a Table of Contents

In this Video Example, you will see how to build a table of contents for a book.

CHAPTER SUMMARY

Understand the architecture of the Java collections framework.

- A collection groups together elements and allows them to be retrieved later.
- A list is a collection that remembers the order of its elements.
- A set is an unordered collection of unique elements.
- A map keeps associations between key and value objects.

Understand and use linked lists.

- A linked list consists of a number of nodes, each of which has a reference to the next node.
- Adding and removing elements at a given position in a linked list is efficient.

Available online in WileyPLUS and at www.wiley.com/college/horstmann.

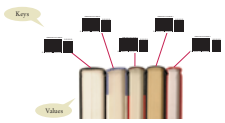
- Visiting the elements of a linked list in sequential order is efficient, but random access is not.
- You use a list iterator to access elements inside a linked list.

Choose a set implementation and use it to manage sets of values.

- The `HashSet` and `TreeSet` classes both implement the `Set` interface.
- Set implementations arrange the elements so that they can locate them quickly.
- You can form hash sets holding objects of type `String`, `Integer`, `Double`, `Point`, `Rectangle`, or `Color`.

- You can form tree sets for any class that implements the `Comparable` interface, such as `String` or `Integer`.
- Sets don't have duplicates. Adding a duplicate of an element that is already present is ignored.
- A set iterator visits the elements in the order in which the set implementation keeps them.
- You cannot add an element to a set at an iterator position.

Use maps to model associations between keys and values.



- The `HashMap` and `TreeMap` classes both implement the `Map` interface.
- To find all keys and values in a map, iterate through the key set and find the values that correspond to the keys.
- A hash function computes an integer value from an object.
- A good hash function minimizes *collisions*—identical hash codes for different objects.

- Override `hashCode` methods in your own classes by combining the hash codes for the instance variables.
- A class's `hashCode` method must be compatible with its `equals` method.

Use the Java classes for stacks, queues, and priority queues.

- A stack is a collection of elements with "last-in, first-out" retrieval.
- A queue is a collection of elements with "first-in, first-out" retrieval.
- When removing an element from a priority queue, the element with the most urgent priority is retrieved.

Solve programming problems using stacks and queues.

- A stack can be used to check whether parentheses in an expression are balanced.
- Use a stack to evaluate expressions in reverse Polish notation.
- Using two stacks, you can evaluate expressions in standard algebraic notation.
- Use a stack to remember choices you haven't yet made so that you can backtrack to them.

STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

java.util.Collection<E>
- add
- contains
- iterator
- remove
- size

java.util.HashMap<K, V>
java.util.HashSet<K, V>
java.util.Iterator<E>
- hasNext
- next
- remove

java.util.LinkedList<E>
- addFirst
- addLast
- getFirst
- getLast
- removeFirst
- removeLast

java.util.List<E>
- listIterator

java.util.ListIterator<E>
- add
- hasPrevious
- previous
- set

java.util.Map<K, V>
- get
- keySet
- put
- remove

java.util.Queue<E>
- peek

java.util.PriorityQueue<E>
- remove

java.util.Set<E>
java.util.Stack<E>
- peek
- pop
- push

java.util.TreeMap<K, V>
java.util.TreeSet<K, V>

REVIEW EXERCISES

■■ **R15.1** An invoice contains a collection of purchased items. Should that collection be implemented as a list or set? Explain your answer.

■■ **R15.2** Consider a program that manages an appointment calendar. Should it place the appointments into a list, stack, queue, or priority queue? Explain your answer.

■■■ **R15.3** One way of implementing a calendar is as a map from date objects to event objects. However, that only works if there is a single event for a given date. How can you use another collection type to allow for multiple events on a given date?

■ **R15.4** Explain what the following code prints. Draw a picture of the linked list after each step.

```
LinkedList<String> staff = new LinkedList<String>();
staff.addFirst("Harry");
staff.addFirst("Diana");
staff.addFirst("Tom");
System.out.println(staff.removeFirst());
System.out.println(staff.removeFirst());
System.out.println(staff.removeFirst());
```

■ **R15.5** Explain what the following code prints. Draw a picture of the linked list after each step.

```
LinkedList<String> staff = new LinkedList<String>();
staff.addFirst("Harry");
staff.addFirst("Diana");
staff.addFirst("Tom");
System.out.println(staff.removeLast());
System.out.println(staff.removeFirst());
System.out.println(staff.removeLast());
```

■ **R15.6** Explain what the following code prints. Draw a picture of the linked list after each step.

```
LinkedList<String> staff = new LinkedList<String>();
staff.addFirst("Harry");
staff.addLast("Diana");
staff.addFirst("Tom");
System.out.println(staff.removeLast());
System.out.println(staff.removeFirst());
System.out.println(staff.removeLast());
```

■ **R15.7** Explain what the following code prints. Draw a picture of the linked list and the iterator position after each step.

```
LinkedList<String> staff = new LinkedList<String>();
ListIterator<String> iterator = staff.listIterator();
iterator.add("Tom");
iterator.add("Diana");
iterator.add("Harry");
iterator = staff.listIterator();
if (iterator.next().equals("Tom")) { iterator.remove(); }
while (iterator.hasNext())  { System.out.println(iterator.next()); }
```

■ **R15.8** Explain what the following code prints. Draw a picture of the linked list and the iterator position after each step.

```
LinkedList<String> staff = new LinkedList<String>();
ListIterator<String> iterator = staff.listIterator();
iterator.add("Tom");
iterator.add("Diana");
iterator.add("Harry");
iterator = staff.listIterator();
iterator.next();
iterator.next();
iterator.add("Romeo");
iterator.next();
iterator.add("Juliet");
iterator = staff.listIterator();
iterator.next();
iterator.remove();
while (iterator.hasNext()) { System.out.println(iterator.next()); }
```

■■ **R15.9** What advantages do linked lists have over arrays? What disadvantages do they have?

■■ **R15.10** Suppose you need to organize a collection of telephone numbers for a company division. There are currently about 6,000 employees, and you know that the phone switch can handle at most 10,000 phone numbers. You expect several hundred look-ups against the collection every day. Would you use an array list or a linked list to store the information?

■■ **R15.11** Suppose you need to keep a collection of appointments. Would you use a linked list or an array list of `Appointment` objects?

■ **R15.12** Suppose you write a program that models a card deck. Cards are taken from the top of the deck and given out to players. As cards are returned to the deck, they are placed on the bottom of the deck. Would you store the cards in a stack or a queue?

■ **R15.13** Suppose the strings `"A"` . . . `"Z"` are pushed onto a stack. Then they are popped off the stack and pushed onto a second stack. Finally, they are all popped off the second stack and printed. In which order are the strings printed?

■ **R15.14** What is the difference between a set and a map?

■■ **R15.15** The union of two sets *A* and *B* is the set of all elements that are contained in *A*, *B*, or both. The intersection is the set of all elements that are contained in *A* and *B*. How can you compute the union and intersection of two sets, using the `add` and `contains` methods, together with an iterator?

■■ **R15.16** How can you compute the union and intersection of two sets, using some of the methods that the `java.util.Set` interface provides, but without using an iterator? (Look up the interface in the API documentation.)

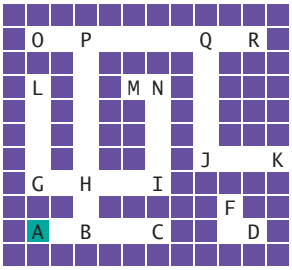
■ **R15.17** Can a map have two keys with the same value? Two values with the same key?

■■ **R15.18** A map can be implemented as a set of (*key*, *value*) pairs. Explain.

■■■ **R15.19** Verify the hash code of the string `"Juliet"` in Table 6.

■■■ **R15.20** Verify that the strings `"VII"` and `"Ugh"` have the same hash code.

■ **R15.21** Consider the algorithm for traversing a maze from Section 15.6.4 Assume that we start at position A and push in the order West, South, East, and North. In which order will the lettered locations of the sample maze be visited?



■ **R15.22** Repeat Exercise R15.21, using a queue instead of a stack.

PROGRAMMING EXERCISES

■■ **P15.1** Write a method

```
public static void downsize(LinkedList<String> employeeNames, int n)
```

that removes every `n`th employee from a linked list.

■■ **P15.2** Write a method

```
public static void reverse(LinkedList<String> strings)
```

that reverses the entries in a linked list.

■■ **P15.3** Use a stack to reverse the words of a sentence. Keep reading words until you have a word that ends in a period, adding them onto a stack. When you have a word with a period, pop the words off and print them. Stop when there are no more words in the input. For example, you should turn the input

```
Mary had a little lamb. Its fleece was white as snow.
```

into

```
Lamb little a had mary. Snow as white was fleece its.
```

Pay attention to capitalization and the placement of the period.

■ **P15.4** Your task is to break a number into its individual digits, for example, to turn 1729 into 1, 7, 2, and 9. It is easy to get the last digit of a number n as `n % 10`. But that gets the numbers in reverse order. Solve this problem with a stack. Your program should ask the user for an integer, then print its digits separated by spaces.

■■ **P15.5** A homeowner rents out parking spaces in a driveway during special events. The driveway is a "last-in, first-out" stack. Of course, when a car owner retrieves a vehicle that wasn't the last one in, the cars blocking it must temporarily move to the street so that the requested vehicle can leave. Write a program that models this behavior, using one stack for the driveway and one stack for the street. Use integers as license plate numbers. Positive numbers add a car, negative numbers remove a car, zero stops the simulation. Print out the stack after each operation is complete.

■ **P15.6** Implement a to do list. Tasks have a priority between 1 and 9, and a description. When the user enters the command `add` *priority description*, the program adds a new task. When the user enters `next`, the program removes and prints the most urgent task. The `quit` command quits the program. Use a priority queue in your solution.

■ **P15.7** Write a program that reads text from a file and breaks it up into individual words. Insert the words into a tree set. At the end of the input file, print all words, followed by the size of the resulting set. This program determines how many unique words a text file has.

■■ **P15.8** Implement the *sieve of Eratosthenes:* a method for computing prime numbers, known to the ancient Greeks. This method will compute all prime numbers up to n. Choose an n. First insert all numbers from 2 to n into a set. Then erase all multiples of 2 (except 2); that is, 4, 6, 8, 10, 12, ... Erase all multiples of 3; that is, 6, 9, 12, 15, ... Go up to $\sqrt{n}$. Then print the set.

■■ **P15.9** Write a program that keeps a map in which both keys and values are strings—the names of students and their course grades. Prompt the user of the program to add or remove students, to modify grades, or to print all grades. The printout should be sorted by name and formatted like this:

```
Carl: B+
Joe: C
Sarah: A
```

■■■ **P15.10** Reimplement Exercise P15.9 so that the keys of the map are objects of class `Student`. A student should have a first name, a last name, and a unique integer ID. For grade changes and removals, lookup should be by ID. The printout should be sorted by last name. If two students have the same last name, then use the first name as a tie breaker. If the first names are also identical, then use the integer ID. *Hint:* Use two maps.

■■■ **P15.11** Write a class `Polynomial` that stores a polynomial such as

$$p(x) = 5x^{10} + 9x^7 - x - 10$$

as a linked list of terms. A term contains the coefficient and the power of x. For example, you would store $p(x)$ as

$$(5,10),(9,7),(-1,1),(-10,0)$$

Supply methods to add, multiply, and print polynomials. Supply a constructor that makes a polynomial from a single term. For example, the polynomial `p` can be constructed as

```
Polynomial p = new Polynomial(new Term(-10, 0));
p.add(new Polynomial(new Term(-1, 1)));
p.add(new Polynomial(new Term(9, 7)));
p.add(new Polynomial(new Term(5, 10)));
```

Then compute $p(x) \times p(x)$.

```
Polynomial q = p.multiply(p);
q.print();
```

■■■ **P15.12** Repeat Exercise P15.11, but use a `Map<Integer, Double>` for the coefficients.

■ **P15.13** Insert all words from a large file (such as the novel "War and Peace", which is available on the Internet) into a hash set and a tree set. Time the results. Which data structure is faster?

■■■ **P15.14** Write a program that reads a Java source file and produces an index of all identifiers in the file. For each identifier, print all lines in which it occurs. For simplicity, we will consider each string consisting only of letters, numbers, and underscores an identifer. Declare a `Scanner in` for reading from the source file and call `in.useDelimiter("[^A-Za-z0-9_]+")`. Then each call to `next` returns an identifier.

■■ **P15.15** Try to find two words with the same hash code in a large file. Keep a `Map<Integer, HashSet<String>>`. When you read in a word, compute its hash code h and put the word in the set whose key is h. Then iterate through all keys and print the sets whose size is > 1.

■■ **P15.16** Supply compatible `hashCode` and `equals` methods to the `Student` class described in Exercise P15.10. Test the hash code by adding `Student` objects to a hash set.

■ **P15.17** Supply compatible `hashCode` and `equals` methods to the `BankAccount` class of Chapter 8. Test the `hashCode` method by printing out hash codes and by adding `BankAccount` objects to a hash set.

■■ **P15.18** A labeled point has x- and y-coordinates and a string label. Provide a class `LabeledPoint` with a constructor `LabeledPoint(int x, int y, String label)` and `hashCode` and `equals` methods. Two labeled points are considered the same when they have the same location and label.

■■ **P15.19** Reimplement the `LabeledPoint` class of the preceding exercise by storing the location in a `java.awt.Point` object. Your `hashCode` and equals methods should call the `hashCode` and `equals` methods of the `Point` class.

■■ **P15.20** Modify the `LabeledPoint` class of Exercise P15.18 so that it implements the `Comparable` interface. Sort points first by their x-coordinates. If two points have the same x-coordinate, sort them by their y-coordinates. If two points have the same x- and y-coordinates, sort them by their label. Write a tester program that checks all cases by inserting points into a `TreeSet`.

■ **P15.21** Write a program that checks whether a sequence of HTML tags is properly nested. For each opening tag, such as `<p>`, there must be a closing tag `</p>`. A tag such as `<p>` may have other tags inside, for example

```
<p> <ul> <li> </li> </ul> <a> </a> </p>
```

The inner tags must be closed before the outer ones. Your program should process a file containing tags. For simplicity, assume that the tags are separated by spaces, and that there is no text inside the tags.

■ **P15.22** Add a `%` (remainder) operator to the expression calculator of Section 15.6.3.

■■ **P15.23** Add a `^` (power) operator to the expression calculator of Section 15.6.3. For example, 2 ^ 3 evaluates to 8. As in mathematics, your power operator should be evaluated from the right. That is, 2 ^ 3 ^ 2 is 2 ^ (3 ^ 2), not (2 ^ 3) ^ 2. (That's more useful because you could get the latter as 2 ^ (3 × 2).)

■■■ **P15.24** Modify the expression calculator of Section 15.6.3 to convert an expression into reverse Polish notation. *Hint:* Instead of evaluating the top and pushing the result, append the instructions to a string.

■■■ **P15.25** Modify the maze solver program of Section 15.6.4 to handle mazes with cycles. Keep a set of visited intersections. When you have previously seen an intersection, treat it as a dead end and do not add paths to the stack.

■■■ **P15.26** In a paint program, a "flood fill" fills all empty pixels of a drawing with a given color, stopping when it reaches occupied pixels. In this exercise, you will implement a simple variation of this algorithm, flood-filling a 10 × 10 array of integers that are initially `0`.

```
Prompt for the starting row and column.
Push the (row, column) pair onto a stack.
```

You will need to provide a simple `Pair` class.

Repeat the following operations until the stack is empty.

```
Pop off the (row, column) pair from the top of the stack.
If it has not yet been filled, fill the corresponding array location with a number 1, 2, 3, and so on
   (to show the order in which the square is filled).
Push the coordinates of any unfilled neighbors in the north, east, south, or west direction on the stack.
```

When you are done, print the entire array.

■ **P15.27** Repeat Exercise P15.26, but use a queue instead.

■■ **P15.28** Use a stack to enumerate all permutations of a string. Suppose you want to find all permutations of the string `meat`.

```
Push the string +meat on the stack.
While the stack is not empty
   Pop off the top of the stack.
   If that string ends in a + (such as tame+)
      Remove the + and add the string to the list of permutations.
   Else
      Remove each letter in turn from the right of the +.
      Insert it just before the +.
      Push the resulting string on the stack.
```

For example, after popping `e+mta`, you push `em+ta`, `et+ma`, and `ea+mt`.

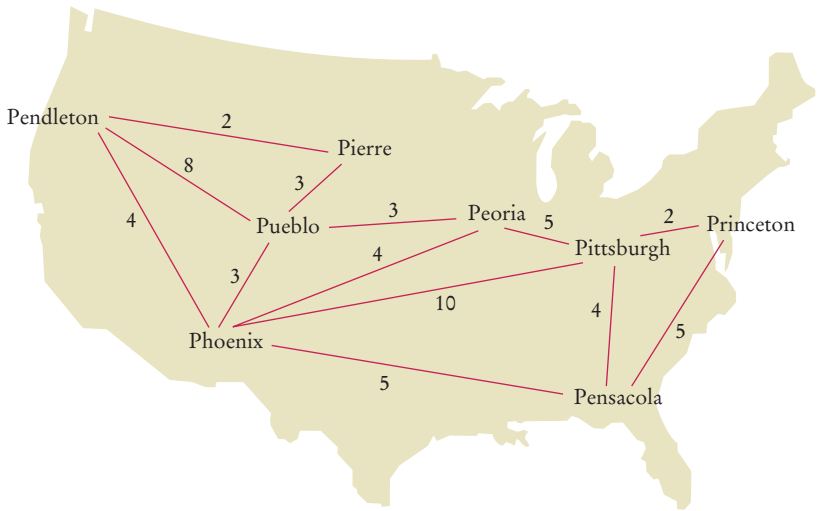
■■ **P15.29** Repeat Exercise P15.28, but use a queue instead.

■■ **Business P15.30** An airport has only one runway. When it is busy, planes wishing to take off or land have to wait. Implement a simulation, using two queues, one each for the planes waiting to take off and land. Landing planes get priority. The user enters commands `takeoff` *flightSymbol*, `land` *flightSymbol*, `next`, and `quit`. The first two commands place the flight in the appropriate queue. The `next` command finishes the current takeoff or landing and enables the next one, printing the action (takeoff or land) and the flight symbol.

■■ **Business P15.31** Suppose you buy 100 shares of a stock at $12 per share, then another 100 at $10 per share, and then sell 150 shares at $15. You have to pay taxes on the gain, but exactly what is the gain? In the United States, the FIFO rule holds: You first sell all shares of the first batch for a profit of $300, then 50 of the shares from the second batch, for a profit of $250, yielding a total profit of $550. Write a program that can make these calculations for arbitrary purchases and sales of shares in a single company. The user enters commands `buy` *quantity price*, `sell` *quantity* (which causes the gain to be displayed), and `quit`. *Hint:* Keep a queue of objects of a class `Block` that contains the quantity and price of a block of shares.

■■■ **Business P15.32** Extend Exercise P15.31 to a program that can handle shares of multiple companies. The user enters commands `buy` *symbol quantity price* and `sell` *symbol quantity*. *Hint:* Keep a `Map<String, Queue<Block>>` that manages a separate queue for each stock symbol.

■■■ **Business P15.33** Consider the problem of finding the least expensive routes to all cities in a network from a given starting point.



For example, in this network, the least expensive route from Pendleton to Peoria has cost 8 (going through Pierre and Pueblo).

The following helper class expresses the distance to another city:

```
public class DistanceTo implements Comparable<DistanceTo>
{
   private String target;
   private int distance;
```

```
   public DistanceTo(String city, int dist) { target = city; distance = dist; }
   public String getTarget() { return target; }
   public int getDistance() { return distance; }
   public int compareTo(DistanceTo other) { return distance - other.distance; }
}
```

All direct connections between cities are stored in a `Map<String, TreeSet<DistanceTo>>`. The algorithm now proceeds as follows:

```
Let from be the starting point.
Add DistanceTo(from, 0) to a priority queue.
Construct a map shortestKnownDistance from city names to distances.
While the priority queue is not empty
   Get its smallest element.
   If its target is not a key in shortestKnownDistance
      Let d be the distance to that target.
      Put (target, d) into shortestKnownDistance.
      For all cities c that have a direct connection from target
         Add DistanceTo(c, d + distance from target to c) to the priority queue.
```

When the algorithm has finished, **shortestKnownDistance** contains the shortest distance from the starting point to all reachable targets.

Your task is to write a program that implements this algorithm. Your program should read in lines of the form *city1 city2 distance*. The starting point is the first city in the first line. Print the shortest distances to all other cities.

ANSWERS TO SELF-CHECK QUESTIONS

1. A list is a better choice because the application will want to retain the order in which the quizzes were given.
2. A set is a better choice. There is no intrinsically useful ordering for the students. For example, the registrar's office has little use for a list of all students by their GPA. By storing them in a set, adding, removing, and finding students can be fast.
3. With a stack, you would always read the latest required reading, and you might never get to the oldest readings.
4. A collection stores elements, but a map stores associations between elements.
5. Yes, for two reasons. A linked list needs to store the neighboring node references, which are not needed in an array, Moreover, there is some overhead for storing an object. In a linked list, each node is a separate object that incurs this overhead, whereas an array is a single object.
6. We can simply access each array element with an integer index.
7. `|ABCD`

   `A|BCD`

   `AB|CD`

   `A|CD`

   `AC|D`

   `ACE|D`

   `ACED|`

   `ACEDF|`
8. ```
   ListIterator<String> iter = words.iterator();
   while (iter.hasNext())
   {
      String str = iter.next();
      if (str.length() < 4) { iter.remove(); }
   }
   ```
9. ```
   ListIterator<String> iter = words.iterator();
   while (iter.hasNext())
   {
      System.out.println(iter.next());
      if (iter.hasNext())
      {
         iter.next(); // Skip the next element
      }
   }
   ```

10. Adding and removing elements as well as testing for membership is faster with sets.
11. Sets do not have an ordering, so it doesn't make sense to add an element at a particular iterator position, or to traverse a set backward.
12. You do not know in which order the set keeps the elements.
13. Here is one possibility:

```
if (s.size() == 3 && s.contains("Tom")
      && s.contains("Diana")
      && s.contains("Harry"))
   . . .
```

14.
```
for (String str : s)
{
   if (t.contains(str))
   {
      System.out.println(str);
   }
}
```

15. The words would be listed in sorted order.
16. A set stores elements. A map stores associations between keys and values.
17. The ordering does not matter, and you cannot have duplicates.
18. Because it might have duplicates.
19. `Map<String, Integer> wordFrequency;`

    Note that you cannot use a `Map<String, int>` because you cannot use primitive types as type parameters in Java.
20. It associates strings with sets of strings. One application would be a thesaurus that lists synonyms for a given word. For example, the key `"improve"` might have as its value the set `["ameliorate", "better", "enhance", "enrich", "perfect", "refine"]`.
21. This way, we can ensure that only queue operations can be invoked on the `q` object.
22. Depending on whether you consider the `0` position the head or the tail of the queue, you would either add or remove elements at that position. Both are expensive operations.
23. `A B C`
24. Stacks use a "last-in, first-out" discipline. If you are the first one to submit a print job and lots of people add print jobs before the printer has a chance to deal with your job, they get their printouts first, and you have to wait until all other jobs are completed.
25. Yes—the smallest string (in lexicographic ordering) is removed first. In the example, that is the string starting with 1, then the string starting with 2, and so on. However, the scheme breaks down if a priority value exceeds 9. For example, a string `"10 - Line up braces"` comes before `"2 - Order cleaning supplies"` in lexicographic order.
26. 70.
27. It would then subtract the first argument from the second. Consider the input 5 3 –. The stack contains 5 and 3, with the 3 on the top. Then `results.pop() - results.pop()` computes 3 – 5.
28. The – gets executed first because + doesn't have a higher precedence.
29. No, because there may be parentheses on the stack. The parentheses separate groups of operators, each of which is in increasing precedence.
30. A B E F G D C K J N