Chapter 3

Testing and Debugging

Chapter Objectives

- To understand different testing strategies and when and how they are performed
- To introduce testing using the JUnit Platform
- To show how to write classes using test-driven development
- To illustrate how to use a debugger within a Java Integrated Development Environment (IDE)

This chapter introduces and illustrates some testing and debugging techniques. We begin by discussing testing in some detail. You will learn how to generate a proper test plan and the differences between unit and integration testing as they apply to an object-oriented design (OOD). Next, we describe the JUnit Platform, which has become a commonly used tool for creating and running unit tests. We also describe test-driven development, a design approach in which the tests and programs are developed in parallel. Finally, we illustrate the debugger, which allows you to suspend program execution at a specified point and examine the value of variables to assist in isolating errors.

Testing and Debugging

3.1 Types of Testing

Testing is the process of exercising a program (or part of a program) under controlled conditions and verifying that the results are as expected. The purpose of testing is to detect program defects after all syntax errors have been removed and the program compiles successfully. The more thorough the testing, the greater the likelihood that the defects will be found. However, no amount of testing can guarantee the absence of defects in sufficiently complex programs. The number of test cases required to test all possible inputs and states that each method may execute can quickly become prohibitively large. That is often why commercial software products have different versions or patches that the user must install. Version n usually corrects the defects that were still present in version $n - 1$.

Testing is generally done at the following levels and in the sequence shown below:

- Unit testing refers to testing the smallest testable piece of the software. In OOD, the unit will be either a method or a class. The complexity of a method determines whether it should be tested as a separate unit or whether it can be tested as part of its class.
- Integration testing involves testing the interactions among units. If the unit is the method, then integration testing includes testing interactions among methods within a class. However, generally it involves testing interactions among several classes.
- System testing is the testing of the whole program in the context in which it will be used. A program is generally part of a collection of other programs and hardware, called a *system*. Sometimes, a program will work correctly until some other software is loaded onto the system and then it will fail for no apparent reason.
- Acceptance testing is system testing designed to show that the program meets its functional requirements. It generally involves use of the system in the real environment or as close to the real environment as possible.

There are two types of testing:

1. *Black-box testing* tests the item (method, class, or program) based on its interfaces and functional requirements. This is also called closed-box testing or functional testing. For testing a method, the input parameters are varied over their allowed range and the results compared against independently calculated results. In addition, values outside the allowed range are tested to ensure that the method responds as specified (e.g., throws an exception or computes a nominal value). Also, the inputs to a method are not only the parameters of the method but also the values of any global data that the method accesses.
2. *White-box testing* tests the software element (method, class, or program) with the knowledge of its internal structure. Other terms used for this type of testing are glass-box testing, open-box testing, and coverage testing. The goal is to exercise as many paths through the element as possible or practical. There are various degrees of coverage. The simplest is statement coverage, which ensures that each statement is executed at least once. Branch coverage ensures that every choice at each branch (`if` statements, `switch` statements, and loops) is tested. For example, if there are only `if` statements, and they are not nested, then each `if` statement is tried with its condition true and with its condition false. This could possibly be done with two test cases: one with all of the `if` conditions true and the other with all of them false. Path coverage tests each path through a method. If there are n `if` statements, path coverage could require 2^n test cases if the `if` statements are not nested (each condition has two possible values, so there could be 2^n possible paths).

EXAMPLE 3.1 Method `testMethod` has a nested `if` statement and displays one of four messages, `path 1` through `path 4`, depending on which path is followed. The values passed to its arguments determine the path. The ellipses represent the other statements in each path.

```
public void testMethod(char a, char b) {
    if (a < 'M') {
        if (b < 'X') {
            System.out.println("path 1");
            ...
        } else {
            System.out.println("path 2");
            ...
        }
    } else {
        if (b < 'C') {
            System.out.println("path 3");
            ...
        } else {
            System.out.println("path 4");
            ...
        }
    }
}
```

To test this method, we need to pass values for its arguments that cause it to follow the different paths. Table 3.1 shows some possible values and the corresponding path.

The values chosen for `a` and `b` in Table 3.1 are the smallest and largest uppercase letters. For a more thorough test, you should see what happens when `a` and `b` are passed values that are between `'A'` and `'Z'`. For example, what happens if the value changes from `'L'` to `'M'`? We pick those values because the condition (`a < 'M'`) has different values for each of them.

Also, what happens when `a` and `b` are not uppercase letters? For example, if `a` and `b` are both digit characters (e.g., `'2'`), the `path 1` message should be displayed because the digit characters precede the uppercase letters (see Appendix A, Table A.2). If `a` and `b` are both lowercase letters, the `path 4` message should be displayed (Why?). If `a` is a digit and `b` is a lowercase letter, the `path 2` message should be displayed (Why?). As you can see, the number of test cases required to test even a simple method such as `testMethod` thoroughly can become quite large.

TABLE 3.1
Testing All Paths of `testMethod`

a	b	Message
'A'	'A'	path 1
'A'	'Z'	path 2
'Z'	'A'	path 3
'Z'	'Z'	path 4

Preparations for Testing

Although testing is usually done after each unit of the software is coded, a test plan should be developed early in the design stage. Some aspects of a test plan include deciding how the software will be tested, when the tests will occur, who will do the testing, and what test data will be used. If the test plan is developed early in the design stage, testing can take place concurrently with the design and coding. Again, the earlier an error is detected, the easier and less expensive it is to correct it.

Another advantage of deciding on the test plan early is that this will encourage programmers to prepare for testing as they write their code. A good programmer will practice defensive programming and include code that detects unexpected or invalid data values. For example, if the parameter `n` for a method is required to be greater than zero, you can place the `if` statement

```
if (n <= 0)
    throw new IllegalArgumentException("n <= 0: " + n);
```

at the beginning of the method. This `if` statement will throw an exception and provide a diagnostic message in the event that the parameter passed to the method is invalid. Method exit will occur and the exception can be handled by the method caller.

Testing Tips for Program Systems

Most of the time, you will be testing program systems that contain collections of classes, each with several methods. Next, we provide a list of testing tips to follow in writing these methods.

1. Carefully document each method parameter and class attribute using comments as you write the code. Also, describe the method operation using comments, following the Javadoc conventions discussed in Appendix Section A.7.
2. Leave a trace of execution by displaying the method name as you enter it.
3. Display the values of all input parameters upon entry to a method. Also, display the values of any class attributes that are accessed by this method. Check that these values make sense.
4. Display the values of all method outputs after returning from a method. Also, display any class attributes that are modified by this method. Verify that these values are correct by hand computation.

You should plan for testing as you write each module rather than after the fact. Include the output statements required for Steps 2 and 3 in the original Java code for the method. When you are satisfied that the method works as desired, you can "remove" the testing statements. One efficient way to remove them is to enclose them in an `if (TESTING)` block as follows:

```
if (TESTING) {
     // Code that you wish to "remove"
     . . .
}
```

You would then define `TESTING` at the beginning of the class as `true` to enable testing,

```
private static final boolean TESTING = true;
```

or as `false` to disable testing,

```
private static final boolean TESTING = false;
```

If you need, you can define different `boolean` flags for different kinds of tests.

EXERCISES FOR SECTION 3.1

SELF-CHECK

1. During which phase of testing would each of the following tests be performed?
   a. Testing whether a method worked properly at all its boundary conditions.
   b. Testing whether class `A` can use class `B` as a component.
   c. Testing whether a phone directory application and a word-processing application can run simultaneously on a personal computer.
   d. Testing whether a method `search` can search an array that was returned by method `buildArray` that stores input data in the array.
   e. Testing whether a class with an array data field can use a static method `search` defined in a class `ArraySearch`.
2. Explain why a method that does not match its declaration in the interface would not be discovered during white-box testing.

3.2 Specifying the Tests

In this section, we discuss how to specify the tests needed to test a program system and its components. The test data may be specified during the analysis and design phases. This should be done for the different levels of testing: unit, integration, and system. In black-box testing, we are concerned with the relationship between the unit inputs and outputs. There should be test data to check for all expected inputs as well as unanticipated data. The test plan should also specify the expected unit behavior and outputs for each set of input data.

In white-box testing, we are concerned with exercising alternative paths through the code. Thus, the test data should be designed to ensure that all `if` statement conditions will evaluate to both `true` and `false`. For nested `if` statements, test different combinations of `true` and `false` values. For `switch` statements, make sure that the selector variable can take on all values listed as case labels and some that are not.

For loops, verify that the result is correct if an immediate exit occurs (zero repetitions). Also, verify that the result is correct if only one iteration is performed and if the maximum number of iterations is performed. Finally, verify that loop repetition can always terminate.

Testing Boundary Conditions

When hand-tracing through an algorithm using white-box testing, you must exercise all paths through the algorithm. It is also important to check special cases called boundary conditions to make sure that the algorithm works for these cases as well as the more common ones. For example, if you are testing a method that searches for a particular target element in an array testing, the boundary conditions means that you should make sure that the method works for the following special cases:

- The target element is the first element in the array.
- The target element is the last element in the array.
- The target is somewhere in the middle.

- The target element is not in the array.
- There is more than one occurrence of the target element, and we find the first occurrence.
- The array has only one element and it is not the target.
- The array has only one element and it is the target.
- The array has no elements.

These boundary condition tests would be required in black-box testing too.

EXERCISES FOR SECTION 3.2

SELF-CHECK

1. List two boundary conditions that should be checked when testing method `readInt` below. The second and third parameters represent the upper and lower bounds for a range of valid integers.

```
/** Returns an integer data value within range minN and maxN inclusive
 *  @param scan a Scanner object
 *  @param minN smallest possible value to return
 *  @param maxN largest possible value to return
 *  @return the first value read between minN and maxN
 */
public static int readInt(Scanner scan, int minN, int maxN) {
    if (minN > maxN)
        throw new IllegalArgumentException ("In readInt, minN " + minN
                                            + " not <= maxN " + maxN) ;
    boolean inRange = false; // Assume no valid number read.
    int n = 0;
    while (!inRange) { // Repeat until valid number read.
        System.out.println("Enter an integer from " + minN + " to "
                           + maxN + ": ") ;
        try {
            n = scan.nextInt();
            inRange = (minN <= n & & n <= maxN) ;
        } catch (InputMismatchException ex) {
            scan.nextLine();
            System.out.println("not an integer - try again");
        }
    } // End while
    return n; // n is in range
}
```

2. Devise test data to test the method `readInt` using
   a. white-box testing
   b. black-box testing

PROGRAMMING

1. Write a `search` method with four parameters: the search array, the target, the start subscript, and the finish subscript. The last two parameters indicate the part of the array that should be searched. Your method should catch or throw exceptions where warranted.

3.3 Stubs and Drivers

In this section, we describe two kinds of methods, stubs and drivers, that facilitate testing. We also show how to document the requirements that a method should meet using preconditions and postconditions.

Stubs

Although we want to do unit testing as soon as possible, it may be difficult to test a method or a class that interacts with other methods or classes. The problem is that not all methods and not all classes will be completed at the same time. So if a method in class `A` calls a method defined in class `B` (not yet written), the unit test for class `A` can't be performed without the help of a replacement method for the one in class `B`. The replacement for a method that has not yet been implemented or tested is called a stub. A stub has the same header as the method it replaces, but its body only displays a message indicating that the stub was called.

EXAMPLE 3.2 The following method is a stub for a `void` method `save`. The stub will enable a method that calls `save` to be tested, even though the real method `save` has not been written.

```
/** Stub for method save.
    pre: the initial directory contents are read from a data file.
    post: Writes the directory contents back to a data file.
          The boolean flag modified is reset to false.
 */
public void save() {
    System.out.println("Stub for save has been called");
    modified = false;
}
```

Besides displaying an identification message, a stub can print out the values of the inputs and can assign predictable values (e.g., 0 or 1) to any outputs to prevent execution errors caused by undefined values. Also, if a method is supposed to change the state of a data field, the stub can do so (`modified` is set to `false` by the stub just shown). If a client program calls one or more stubs, the message printed by each stub when it is executed provides a trace of the call sequence and enables the programmer to determine whether the flow of control within the client program is correct.

Preconditions and Postconditions

In the comment for the method `save`, the lines

```
pre: the initial directory contents are read from a data file.
post: Writes the directory contents back to a data file.
      The boolean flag modified is reset to false.
```

show the precondition (following `pre:`) and postcondition (following `post:`) for the method `save`. A *precondition* is a statement of any assumptions or constraints on the method `data` (input parameters) before the method begins execution. A *postcondition* describes the result of executing the method. A method's preconditions and postconditions serve as a contract between a method caller and the method programmer—if a caller satisfies the precondition, the method result should satisfy the postcondition. If the precondition is not satisfied, there is no guarantee that the method will do what is expected, and it may even fail. The preconditions and postconditions allow both a method user and a method implementer to proceed without further coordination.

We will use postconditions to describe the change in object state caused by executing a mutator method. As a general rule, you should write a postcondition comment for all `void` methods. If a method returns a value, you do not usually need a postcondition comment because the `@return` comment describes the effect of executing the method.

Drivers

Another testing tool for a method is a driver program. A driver program declares any necessary object instances and variables, assigns values to any of the method's inputs (as specified in the method's preconditions), calls the method, and displays the values of any outputs returned by the method. Alternatively, driver methods can be written in a separate test class and executed under the control of a test framework such as JUnit, which we discuss in the next section.

EXERCISES FOR SECTION 3.3

SELF-CHECK

1. Can a main method be used as a stub or a driver? Explain your answer.
2. Write a stub to use in place of the method `readInt` in Self-Check Exercise 1 of Section 3.2.

PROGRAMMING

1. Writer a driver program to the test method `readInt` in Self-Check Exercise 1 of Section 3.2 using the test data derived for Self-Check Exercise 2, part b in Section 3.2.
2. Write a driver program to test method `search` in Programming Exercise 1, Section 3.2.

3.4 The JUnit5 Platform

A *test class* is a class that contains one or more *test methods* to test a method or class. A test method provides known inputs and then compares the expected and actual results of a test and an indication of pass or fail.

A *test framework* is a software product that facilitates writing test cases, organizing the test methods, running the tests, and reporting the results. The JUnit5 Platform is currently the latest version of an open-source product that serves as a foundation for a test framework called `jupiter`. It can be used stand-alone or with popular IDEs (e.g., NetBeans, Eclipse, and IntelliJ). In the next section, we show a test class for the `ArraySearch` class constructed using the JUnit5 Platform.

A test class may also contain common code to be executed before/after each test to ensure that the class being tested is in a known state. Test classes will be discovered by the test engine, which will then execute them.

Each test class begins with:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
```

The first `import` statement makes the `Test` interface visible, which allows us to use the `@Test` attribute to identify test cases. *Annotations* such as `@Test` are directions to the compiler and other language-processing tools; they do not affect the execution of the program. The JUnit

test engine searches classes for methods with the `@Test` annotation. When it executes them, it keeps track of the pass/fail results.

The second `import` statement makes the methods in the `Assertions` class visible. The `assert` methods are used to determine pass/fail results for a test. Table 3.2 describes the various `assert` methods that are defined in `org.junit.jupiter.api.Assertions`. If an `assert` method fails, then an exception is thrown causing an error to be reported as specified in the description for method `assertArrayEquals`. If one of the assertions fail, then the test fails; if none fails, then the test passes.

TABLE 3.2
Methods Defined in `org.junit.jupiter.api.Assertions`

Method	Parameters	Description
`assertArrayEquals`	*expected*, *actual*[,*message*]	Tests to see whether the contents of the two array parameters *expected* and *actual* are equal. This method is overloaded for arrays of the primitive types and `Object`. Arrays of `Objects` are tested with the `.equals` method applied to the corresponding elements. The test fails if an unequal pair is found, and an `AssertionError` is thrown. If the optional *message* is included, the `AssertionError` is thrown with this *message* followed by the default message; otherwise it is thrown with a default message
`assertEquals`	*expected*, *actual*[, *message*]	Tests to see whether *expected* and *actual* are equal. This method is overloaded for the primitive types and `Object`. To test `Objects`, the `.equals` method is used
`assertFalse`	*condition*[, *message*]	Tests to see whether the `boolean` expression *condition* is false
`assertNotEquals`	*expected*, *actual*[, *message*]	Tests to see whether *expected* and *actual* are not equal. This method is overloaded for the primitive types and `Object`. To test `Objects`, the `.equals` method is used.
`assertNotNul1`	*object*[, *message*]	Tests to see if the *object* is not `null`
`assertNotSame`	*expected*, *actual*[, *message*]	Tests to see if *expected* and *actual* are not the same object. (Applies the `!=` operator.)
`assertNul1`	*object*[, *message*]	Tests to see whether the *object* is `null`
`assertSame`	*expected*, *actual*[, *message*]	Tests to see whether *expected* and *actual* are the same object. (Applies the `==` operator.)
`assertThrows`	*expected*, *executable* [, *message*]	Tests to see whether the code in the *executable* block throws the expected *exception*. (Explained below)
`assertTrue`	*condition*[, *message*]	Tests to see whether the `boolean` expression *condition* is true
`fail`	[*message*]	Always throws `AssertionError`

EXAMPLE 3.3 Listing 3.1 shows a JUnit test class for an array search method (`ArraySearch.search`) that returns the location of the first occurrence of a target value (the second parameter) in an array (the first parameter) or –1 if the target is not found. The test class contains methods that implement the tests first described in Section 3.2 and repeated below.

- The target element is the first element in the array.
- The target element is the last element in the array.
- The target is somewhere in the middle.

- The target element is not in the array.
- There is more than one occurrence of the target element and we find the first occurrence.
- The array has only one element and it is not the target.
- The array has only one element and it is the target.
- The array has no elements.

Method `firstElementTest` in Listing 3.1 implements the first test case. It tests to see whether the target is the first element in the 7-element array {5, 12, 15, 4, 8, 12, 7}. In the statement

```
assertEquals(0, ArraySearch.search(x, 5), "5 is not at position 0");
```

the call to method `ArraySearch.search` returns the location of the target (5) in array `x`. The test passes (`ArraySearch.search` returns 0), and JUnit remembers the result.

The Test Results window shown in Figure 3.1a appears after a test class is run in Eclipse. The check mark before each test method shows that the tests all passed.

If instead we used the following statement that incorrectly searches for target 4 as the first array element

```
assertEquals(0, ArraySearch.search(x, 4), "4 is not at position 0");
```

FIGURE 3.1a
Test Results

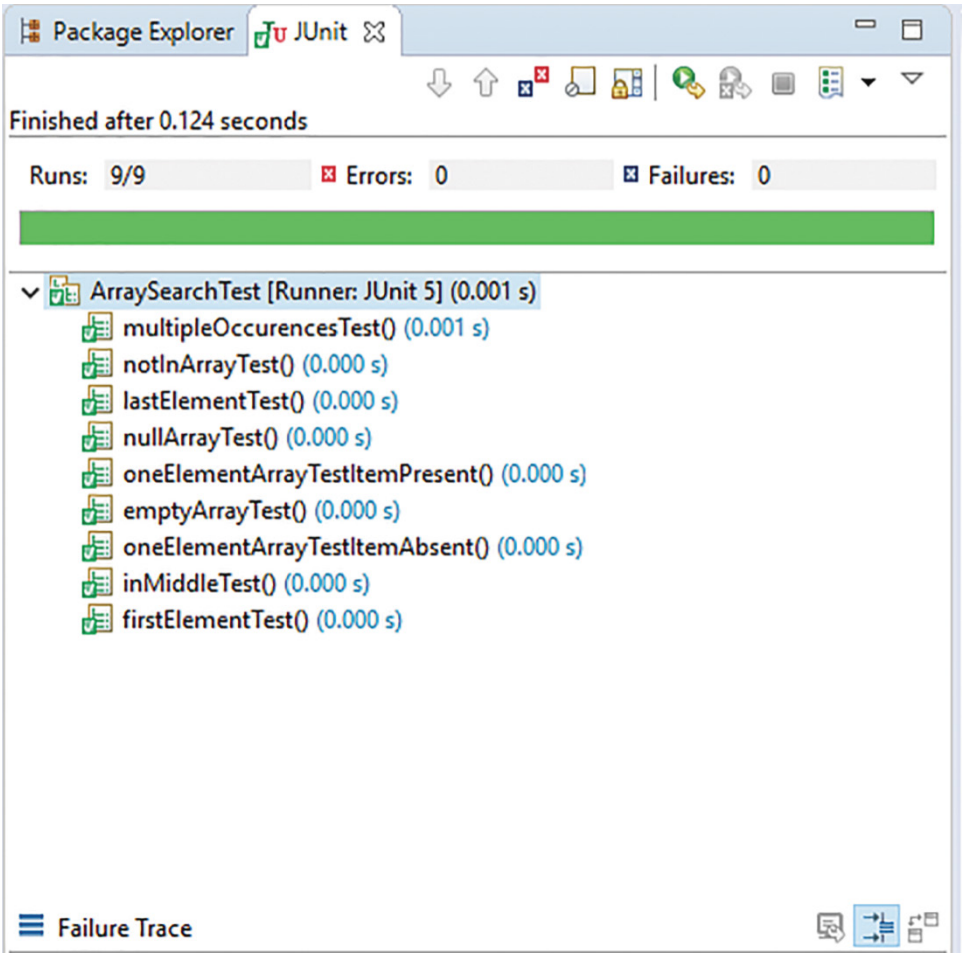
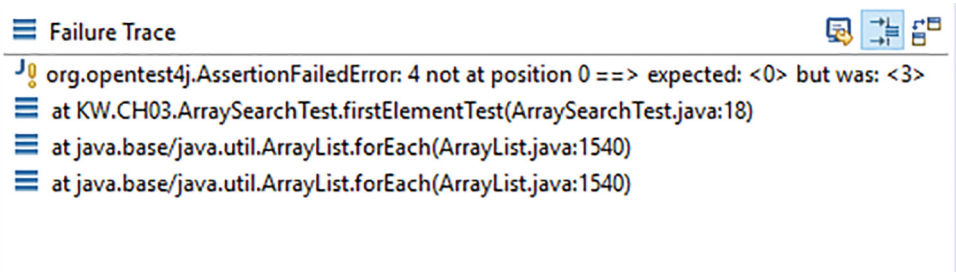


FIGURE 3.1b
Failure Trace



the test would fail and the Test Results window would place an x in front of test method `firstElementTest`. The first line of the Failure Trace window (Figure 3.1b) would show the failure message followed by a trace of the method calls leading to the failure. The gray box at the end of this section shows how to access JUnit5 in Eclipse and IntelliJ.

LISTING 3.1
JUnit test of `ArraySearch.search`

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

/**
 * JUnit test of ArraySearch.search
 * @author Koffman and Wolfgang
 */
public class ArraySearchTest {

    // Common array to search for most of the tests
    private final int[] x = {5, 12, 15, 4, 8, 12, 7};
    @Test
    public void firstElementTest() {
        // Test for target as first element.
        assertEquals(0, ArraySearch.search(x, 5),
                "5 not at position 0");
    }

    @Test
    public void lastElementTest() {
        // Test for target as last element.
        assertEquals(6, ArraySearch.search(x, 7),
                "7 not at position 6");
    }

    @Test
    public void inMiddleTest() {
        // Test for target somewhere in middle.
        assertEquals(3, ArraySearch.search(x, 4),
                "4 is not found at position 3");
    }

    @Test
    public void notInArrayTest() {
        // Test for target not in array.
        assertEquals(-1, ArraySearch.search(x, -5));
    }
```

```
    @Test
    public void multipleOccurrencesTest() {
        // Test for multiple occurrences of target.
        assertEquals(1, ArraySearch.search(x, 12));
    }

    @Test
    public void oneElementArrayTestItemPresent() {
        // Test for 1-element array
       int[] y = {10};
       assertEquals(0, ArraySearch.search(y, 10));
    }

    @Test
    public void oneElementArrayTestItemAbsent() {
        // Test for 1-element array
        int[] y = {10};
        assertEquals(-1, ArraySearch.search(y, -10));
    }
    @Test
    public void emptyArrayTest() {
        // Test for an empty array
        int[] y = new int[0];
        assertEquals(-1, ArraySearch.search(y, 10));
    }

    @Test
    public void nullArrayTest() {
        // Test for a null array
        int[] y = null;
        assertThrows(NullPointerException.class,
            () -> ArraySearch.search(y, 10));
    }
}
```

The last test case in Listing 3.1 does not implement one of the tests in our earlier list of test cases. Its purpose is to test that the `ArraySearch.search` method fails as it should when it is passed a `null` array. To tell JUnit that a test is expected to throw an exception, we use the `assertThrows` method. The `assertThrows` method takes two parameters: the exception class that is expected to be thrown and the code that we expect to throw the exception. The notation `() ->` begins a lambda expression. (We explain lambda expressions in Section 6.4.) In this case, we write the method call that we expect to throw the exception after the characters `() ->`.

JUnit in Eclipse

It is fairly easy to create a JUnit test class in Eclipse. Once you have written class `ArraySearch.java`, select File -> new-> JUnit Test Case. A Create Tests window will pop up. Select OK and then a Select JUnit Version window will pop up: select the most recent version of JUnit (currently JUnit 5). At this point, a new class will be created (`ArraySearchTest.java`) that will contain prototype tests for all the public functions in class `ArraySearch`. You can replace the prototype tests with your own. To execute the tests, select run -> run as -> JUnit Test.

JUnit in IntelliJ

To create a JUnit test class in IntelliJ for the class in an Edit window, click on the class name in the class header and press Alt-Enter. Then select Create Test from the drop-down menu. Press OK if a window with the label *No Test Roots Found* pops up with the question *Create test in the same source root?* Next, a *Create Test* window will appear. The first line of the Create Test window has the label *Testing Library:*, followed by a pull-down menu showing several choices. Select JUnit5 as the Testing Library. The next line should show a light bulb followed by the statement *JUnit5 library not found in the module* and a Fix button. Press Fix and a window will pop up with the label *Download Library from Maven Depository* (Maven is the test engine). The first line will be a pull-down menu with the heading *jupiter:junit-jupiter-api:RELEASE*. A search for possible libraries will begin. When the search finishes, the library choices will appear in the pull-down menu. Select the option *org.junit.jupiter:junit-jupiter-api:5.6.0* from the pull-down menu. Next, check the box before the label *Download to*. The default directory shown is the path to your project with `\lib` appended. Press OK at the bottom of the Dialog window, and the JUnit5 library will be downloaded into the `lib` directory of your project and you will then return to the *Create Test* window. The next line shows the name of the new testing class, which is the name of the class being tested with `Test` appended. Finally, click OK in the Create Test window. The testing class will appear in a new Edit window with `import static org.junit.jupiter.api.Assertions.*;` as the first line. `Assertions` is in color because there are unresolved references to it. Click on `Assertions`, press Alt-Enter, and a drop-down menu will appear. Select the first choice that shows a red light bulb followed by *Add Junit5.4 to classpath*. The project lib directory will then include file `junit-jupiter-api-5.6.0.jar`. and the references to `Assertions` will be resolved. Then, you can type the line `import org.junit.jupiter.api.Test;` and start writing tests in the testing class.

EXERCISES FOR SECTION 3.4

SELF-CHECK

1. List the boundary conditions and tests needed for a method with the following heading:

```
/**
 * Search an array to find the first occurrence of the
 * largest element
 * @param x Array to search
 * @return The subscript of the first occurrence of the
 *         largest element
 * @throws NullPointerException if x is null
 */
public static int findLargest(int[] x) {
```

2. Modify the test(s) in the list for Example 3.3 to verify a search method that finds the last occurrence of a target element in an array.

PROGRAMMING

1. Write the JUnit test class for the method described in Self-Check Exercise 1.
2. Write the JUnit test class for the modification to the search method described in Self-Check Exercise 2.

3.5 Test-Driven Development

Rather than writing a complete method and then testing it, test-driven development involves writing the tests and the method in parallel. The sequence is as follows:

Write a test case for a feature.

Run the test and observe that it fails, but other tests still pass.

Make the minimum change necessary to make the test pass.

Revise the method to remove any duplication between the code and the test.

Rerun the test to see that it still passes.

We then repeat these steps adding a new feature until all of the requirements for the method have been implemented.

We will use this approach to develop a method to find the first occurrence of a target in an array.

Case Study: Test-Driven Development of `ArraySearch.search`

Write a program to search an array that performs the same way as Java method `ArraySearch.search`. This method should return the index of the first occurrence of a target in an array, or –1 if the target is not present.

We start by creating a test list like that in the last section and then work through them one at a time. During this process, we may think of additional tests to add to the test list.

Our test list is as follows:

1. The target element is not in the array.
2. The target element is the first element in the array.
3. The target element is the last element in the array.
4. There is more than one occurrence of the target element and we find the first occurrence.
5. The target is somewhere in the middle.
6. The array has only one element.
7. The array has no elements.

Next we create a stub for the method we want to code:

```
/**
 * Provides a static method search that searches an array
 * @author Koffman & Wolfgang
 */
public class ArraySearch {
    /**
     * Search an array to find the first occurrence of a target
     * @param x Array to search
     * @param target Target to search for
     * @return The subscript of the first occurrence if found:
```

```
     *         otherwise return -1
     * @throws NullPointerException if x is null
     */
    public static int search(int[] x, int target) {
        return Integer.MIN_VALUE;
    }
}
```

Now, we create the first test that combines tests 1 and 6 above. We will screen the test code in blue to distinsguish it from the search method code.

```
/**
 * Test for ArraySearch class
 * @author Koffman & Wolfgang
 */
public class ArraySearchTest {
    @Test
    public void itemNotFirstElementInSingleElementArray() {
        int[] x = {5};
        assertEquals(-1, ArraySearch.search(x, 10));
    }
}
```

And when we run this test, we get the message:

```
Testcase: itemNotFirstElementInSingleElementArray: FAILED
expected:<-1> but was:<-2147483648>
```

The minimum change to enable method search to pass the test is

```
public static int search(int[] x, int target) {
    return -1; // target not found
}
```

Now, we can add a second test to see whether we find the target in the first element (tests 2 and 6 above).

```
@Test
public void itemFirstElementInSingleElementArray() {
    int[] x = {5};
    assertEquals(0, ArraySearch.search(x, 5));
}
```

As expected, this test fails because the search method returns –1. To make it pass, we modify our search method:

```
public static int search(int[] x, int target) {
    if (x[0] == target) {
        return 0; // target found at 0
    }
    return -1; // target not found
}
```

Both tests for a single-element array now pass. Before moving on, let us see whether we can improve this. The process of improving code without changing its functionality is known as *refactoring*. Refactoring is an important step in Test-Driven Development (TDD). It is also facilitated by TDD since having a working test suite gives you the confidence to make changes. (Kent Beck, a proponent of TDD says that TDD gives courage.[1])

The statement:

```
return 0;
```

[1] Beck, Kent. *Test-Driven Development by Example*. Addison-Wesley, 2003.

is a place for possible improvement. The value 0 is the index of the target. For a single-element array, this is obviously 0, but for larger arrays it may be different. Thus, an improved version is

```
public static int search(int[] x, int target) {
    int index = 0;
    if (x[index] == target)
        return index; // target at 0
    return -1; // target not found
}
```

Now, let us see whether we can find an item that is last in a larger array (test 3 above). We start with a 2-element array:

```
@Test
public void itemSecondItemInTwoElementArray() {
    int[] x = {10, 20};
    assertEquals(1, ArraySearch.search(x, 20));
}
```

The first two tests still pass, but the new test fails. As expected, we get the message:

```
Testcase: itemSecondItemInTwoElementArray: FAILED
expected:<1> but was:<-1>
```

The test failed because we did not compare the second array element to the target. We can modify the method to do this as shown next.

```
public static int search(int[] x, int target) {
    int index = 0;
    if (x[index] == target)
        return index; // target at 0
    index = 1;
    if (x[index] == target)
        return index; // target at 1
    return -1; // target not found
}
```

However, this would result in an `ArrayIndexOutOfBoundsException` error for test `itemNotFirstElementInSingleElementArray` because there is no second element in the array `{5}`. If we change the method to first test that there is a second element before comparing it to target, all tests will pass.

```
public static int search(int[] x, int target) {
    int index = 0;
    if (x[index] == target)
        return index; // target at 0
    index = 1;
    if (index < x.length) {
        if (x[index] == target)
            return index; // target at 1
    }
    return -1; // target not found
}
```

However, what happens if we increase the number of elements beyond 2?

```
@Test
public void itemLastInMultiElementArray() {
    int[] x = {5, 10, 15};
    assertEquals(2, ArraySearch.search(x, 15));
}
```

This test would fail because the target is not at position 0 or 1. To make it pass, we could continue to add `if` statements to test more elements, but this is a fruitless approach. Instead, we should modify the code so that the value of `index` advances to the end of the array. We can change the second `if` to a `while` and add a statement to increment `index`.

```
public static int search(int[] x, int target) {
    int index = 0;
    if (x[index] == target)
        return index; // target at 0
    index = 1;
    while (index < x.length) {
       if (x[index] == target)
           return index; // target at index
       index++;
    }
    return -1; // target not found
}
```

At this point, we have a method that will pass all of the tests for any size array. We can group all the tests in a single testing method to verify this.

```
@Test
public void verificationTests() {
    int[] x = {5, 12, 15, 4, 8, 12, 7};
    // Test for target as first element
    assertEquals(0, ArraySearch.search(x, 5));
    // Test for target as last element
    assertEquals(6, ArraySearch.search(x, 7));
    // Test for target not in array
    assertEquals(-1, ArraySearch.search(x, -5));
    // Test for multiple occurrences of target
    assertEquals(1, ArraySearch.search(x, 12));
    // Test for target somewhere in middle
    assertEquals(3, ArraySearch.search(x, 4));
}
```

Although it may look like we are done, we are not finished because we also need to check that an empty array will always return –1:

```
@Test
public void itemNotInEmptyArray() {
    int[] x = {};
    assertEquals(-1, ArraySearch.search(x, 5));
}
```

Unfortunately, this test does not pass because of an `ArrayIndexOutofboundsException: Index 0 out of bounds for length 0` in the first `if` condition for method `search`. If we look closely at the code for `search`, we see that the initial test for when `index` is 0 is the same as for the other elements. So we can remove the first statement and start the loop at 0 instead of 1 (another example of refactoring). Our code becomes more compact and this test will also pass. A slight improvement would be to replace the `while` with a `for` statement.

```
public static int search(int[] x, int target) {
    int index = 0;
    while (index < x.length) {
        if (x[index] == target)
            return index; // target at index
        index++;
     }
     return -1; // target not found
}
```

Finally, if we pass a `null` pointer instead of a reference to an array, a `NullPointerException` should be thrown (an additional test not in our original list).

```
@Test
public void nullValueOfXThrowsException() {
    assertThrows(NullPointerException.class,
        ()-> ArraySearch.search(null, 5));
}
```

EXERCISES FOR SECTION 3.5

SELF-CHECK

1 Why did the first version of method `search` that passed the first test `itemNotFirstElementInSingleElementArray` contain only the statement `return -1`?

2 Assume the first JUnit test for the `findLargest` method described in Self-Check Exercise 2 in Section 3.4 is a test that determines whether the first item in a one element array is the largest. What would be the minimal code for a method `findLargest` that passed this test?

PROGRAMMING

1. Write the `findLargest` method described in Self-Check Exercise 2 in Section 3.4 using Test-Driven Development.

3.6 Testing Interactive Programs in JUnit

In this section, we show how to use JUnit to test a method that gets an integer value in a specified range from the program user. Listing 3.2 shows method `readInt`, which is defined in class `MyInput`.

LISTING 3.2
Method `MyInput.readInt`

```
/**
 * Method to return an integer data value between two
 * specified end points.
 * pre: minN <= maxN
 * @param prompt prompting Message
 * @param minN Smallest value in range
 * @param maxN Largest value in range
 * @throws IllegalArgumentException
 * @return The first data value that is in range
 */
public static int readInt(String prompt, int minN, int maxN) {
    if (minN > maxN) {
        throw new IllegalArgumentException("In readInt, minN " + minN +
                                           "not <= maxN " + maxN);
    }
    // Arguments are valid, read a number
    boolean inRange = false; //Assume no valid number read
    int n = 0;
```

```
    var in = new Scanner(System.in);
    while (!inRange) {
        try {
            System.out.println(prompt + "\nEnter an integer between "
                               + minN + " and " + maxN);
            String line = in.nextLine();
            n = Integer.parseInt(line);
            inRange = (minN <= n && n <= maxN);
        } catch (NumberFormatException ex) {
            System.out.println("Bad numeric string - Try again");
        }
    }
    return n;
}
```

The advantage of using a test framework such as JUnit is that tests are automated. They do not require any user input, and they always present the same test cases to the unit being tested. With JUnit, we can test that an `IllegalArgumentException` is thrown if the input parameters are not valid:

```
@Test
public void testForInvalidInput() {
    Exception ex = assertThrows(IllegalArgumentException.class,
        () -> MyInput.readInt("Enter weight", 5, 2));
    assertEquals("In readInt, minN 5 not <= maxN 2",
        ex.getMessage());
}
```

If the expected exception is thrown, the `assertThrows` method will return the exception object, which can then test to see that the returned message is as expected.

We also need to test that the method works correctly for values that are within a valid range and for values that are outside without requiring the user to enter these data. So we need to provide repeatable user input, and we need to verify that the output displayed is correct.

`System.in` is a public static field in the `System` class. It is initialized to an `InputStream` that reads from the system-defined standard input. The method `System.setIn` can be used to change the value of `System.in`. Similarly, `System.out` is initialized to a `PrintStream` that writes to the operating system-defined standard output, and the method `System.setOut` can be used to change it.

ByteArrayInputStream

The `ByteArrayInputStream` is an `InputStream` that provides input from a fixed array of bytes. The array is initialized by the constructor. Calls to the `readInt` method return successive bytes from the array until the end of the array is reached. Thus, if we wanted to test what `readInt` does when the string `"3"` is entered, we can do the following:

```
@Test
public void testForNormalInput() {
    var testIn = new ByteArrayInputStream("3".getBytes());
    System.setIn(testIn);
    int n = MyInput.readInt("Enter weight", 2, 5);
    assertEquals(n, 3);
}
```

ByteArrayOutputStream

The `ByteArrayOutputStream` is an `OutputStream` that collects each byte written to it into an internal byte array. The `toString` method will then convert the current contents into a

String. To capture and verify output written to System.out, we need to create a PrintStream that writes to a ByteArrayOutputStream and then set System.out to this PrintStream.

To verify that the prompt is properly displayed for the normal case, we use the following test:

```
@Test
public void testThatPromptIsCorrectForNormalInput() {
    var testIn = new ByteArrayInputStream("3".getBytes());
    System.setIn(testIn);
    var testOut = new ByteArrayOutputStream();
    System.setOut(new PrintStream(testOut));
    int n = MyInput.readInt("Enter weight", 2, 5);
    assertEquals(n, 3);
    var displayedPrompt = testOut.toString();
    var expectedPrompt = "Enter weight" +
        "\nEnter an integer between 2 and 5" + NL;
    assertEquals(expectedPrompt, displayedPrompt);
}
```

For the data string "3", the string formed by testOut.toString() should match the string in expectedPrompt. String expectedPrompt ends with the string constant NL instead of \n. This is because the println method ends the output with a system-dependent line terminator. On the Windows operating system, this is the sequence \r\n, and on the Linux operating system it is \n. The statement

```
private static final String NL = System.getProperty("line.separator");
```

initializes the String constant NL to the system-specific line terminator.

Finally, we need to write additional tests (4 in all) that verify that method readInt works properly when an invalid integer string is entered and when the integer string entered is not in range. Besides verifying that the value returned is correct, you should verify the prompts displayed. For each of these tests, your test input should include a valid input following the invalid input so that the method will return normally. These tests are left to Programming Exercises 1 and 2. For example, the statement

```
var testIn = new ByteArrayInputStream("X\n 3".getBytes());
```

would provide a data sample for the first test case ("X" is an invalid integer string, "3" is valid).

EXERCISES FOR SECTION 3.6

SELF-CHECK

1. Explain why it is not necessary to write a test to verify that readInt works properly when the input consists of an invalid integer string, followed by an out-of-range integer, then followed by an integer that is in range.

PROGRAMMING

1. Write separate tests to verify the result returned by readInt and the prompts displayed when the input consists of an invalid integer string followed by a valid integer. Verify that the expected error message is presented followed by a repeat of the prompt.
2. Write separate tests to verify the result returned by readInt and the prompts displayed when the input consists of an out-of-range integer followed by a valid integer. Verify that the expected error message is presented followed by a repeat of the prompt.

3.7 Debugging a Program

In this section, we discuss the process of debugging (removing errors) both with and without the use of a debugger program. Debugging is the major activity performed by programmers during the testing phase. Testing determines whether you have an error; during debugging you determine the cause of run-time and logic errors and correct them, without introducing new ones. If you have followed the suggestions for testing described in the previous section, you will be well prepared to debug your program.

Debugging is like detective work. To debug a program, you must inspect carefully the information displayed by your program, starting at the beginning, to determine whether what you see is what you expect. For example, if the result returned by a method is incorrect but the arguments (if any) passed to the method had the correct values, then there is a problem inside the method. You can try to trace through the method to see whether you can find the source of the error and correct it. If you can't, you may need more information. One way to get that information is to insert additional diagnostic output statements in the method. For example, if the method contains a loop, you may want to display the values of loop control variables during loop execution.

EXAMPLE 3.4 The loop in Listing 3.3 is used to read words entered at the terminal into a list. A sentinel value of `"***"` is used to terminate loop execution and should not be stored in the list. After loop exit the new list is returned to the caller.

Unfortunately, this loop does not terminate when the user enters the sentinel string. The loop exits eventually after the user has entered 10 data items and the sentinel will be saved in the list.

LISTING 3.3
Method `getStrings`

```
/**
 * Return a list filled with the individual words entered by the user.
 * The user can enter the sentinel *** to terminate data entry.
 * @return A list with a maximum of 10 words
 */
public static List<String> getStrings() {
    var in = new Scanner(System.in);
    List<String> stringy = new ArrayList<>();
    int count = 0;
    while (count < 10) {
        System.out.println("Enter a word or *** to quit");
        String word = in.next();
        if (word == "***") break;
        stringy.add(word);
        count++;
    }
    return stringy;
}
```

To determine the source of the problem, you should insert a diagnostic output statement that displays the values of `word` and `count` to make sure that `word` is receiving the sentinel string (`"***"`). You could insert the line

```
System.out.println("!!! Next word is " + word + ", count is " + count);
```

as the first statement in the loop body. If the third data item you enter is the sentinel string, you will get the output line:

```
!!! next word is ***, count is 2
```

This will show you that `word` does indeed receive the sentinel string, but the loop body continues to execute. Therefore, there must be something wrong with the `if` statement that tests for the sentinel. In fact, the `if` statement must be changed to

```
if (word.equals("***")) break;
```

because `word == "***"` compares the address of the string stored in `word` with the address of the literal string `"***"`, not the contents of the two strings as intended. The strings' addresses will always be different, even when their contents are the same. To compare their contents, the `equals` method must be used.

Using a Debugger

If you are using an IDE, you will most likely have a debugger program as part of the IDE. A debugger can execute your program incrementally rather than all at once. After each increment of the program executes, the debugger pauses, and you can view the contents of variables to determine whether the statement(s) executed as expected. You can inspect all the program variables without needing to insert diagnostic output statements. When you have finished examining the program variables, you direct the debugger to execute the next increment.

You can choose to execute in increments as small as one program statement (called *single-step execution*) to see the effect of each statement's execution. Another possibility is to set breakpoints in your program to divide it into sections. The debugger can execute all the statements from one breakpoint to the next as a group. For example, if you wanted to see the effects of a loop's execution but did not want to step through every iteration, you could set breakpoints at the statements just before and just after the loop.

When your program pauses, if the next statement contains a call to a method you have written, you can select single-step execution in the method being called (i.e., `Step Into` the method). If the next statement contains a call to a method in the Java system, you can execute all the method statements as a group and pause after the return from the method execution (i.e., `Step Over` the method). The actual mechanics of using a debugger depends on your IDE. However, the process that you follow is similar among IDEs, and if you understand the process for one, you should be able to use any debugger. Next, we demonstrate how to use the debuggers in IntelliJ and Eclipse.

The IntelliJ and Eclipse Debuggers

Before starting a debugger, you must set a breakpoint so the program's execution can be traced. We will place the breakpoint at the last line of the loop body (line 20) because this will enable us to see the change in list `stringy` at the end of each iteration. In IntelliJ, you move the cursor to the line number of the intended breakpoint and click to the right of the line number. (*In Eclipse, you right-click to the left of the line number.*) In IntelliJ, a red circle next to the line number denotes a breakpoint (*a green circle in Eclipse*). To remove a breakpoint, you click on the circle.

To start the debugger, click the icon shaped like a bug (or) or select Run -> Debug from the main menu. When method `getStrings` is called, the loop will be entered, and the program will pause for data entry at the statement that enters a string. You type in a data word and press Enter. The program then resumes execution until the breakpoint at line 20 is reached and the program is suspended.

To trace the loop's execution, we need to begin single-step execution. In IntelliJ, you begin single statement execution by clicking the icon at the top of the Console window that resembles a crooked arrow ( ) or F8 or by selecting Step Over from Run -> Step Over. (*In Eclipse, you click on the icon ( ) or F6 or select Run -> Step Over*).

A color bar highlights the next statement to be executed. Figure 3.2 shows the Edit window in IntelliJ when the breakpoint is reached for the third time. In IntelliJ, the size (3) of `stringy` and the contents of `word` (`"***"`) appear to the right of the statement at the breakpoint.

FIGURE 3.2 Edit Window (top right) and Console Window (bottom) in IntelliJ

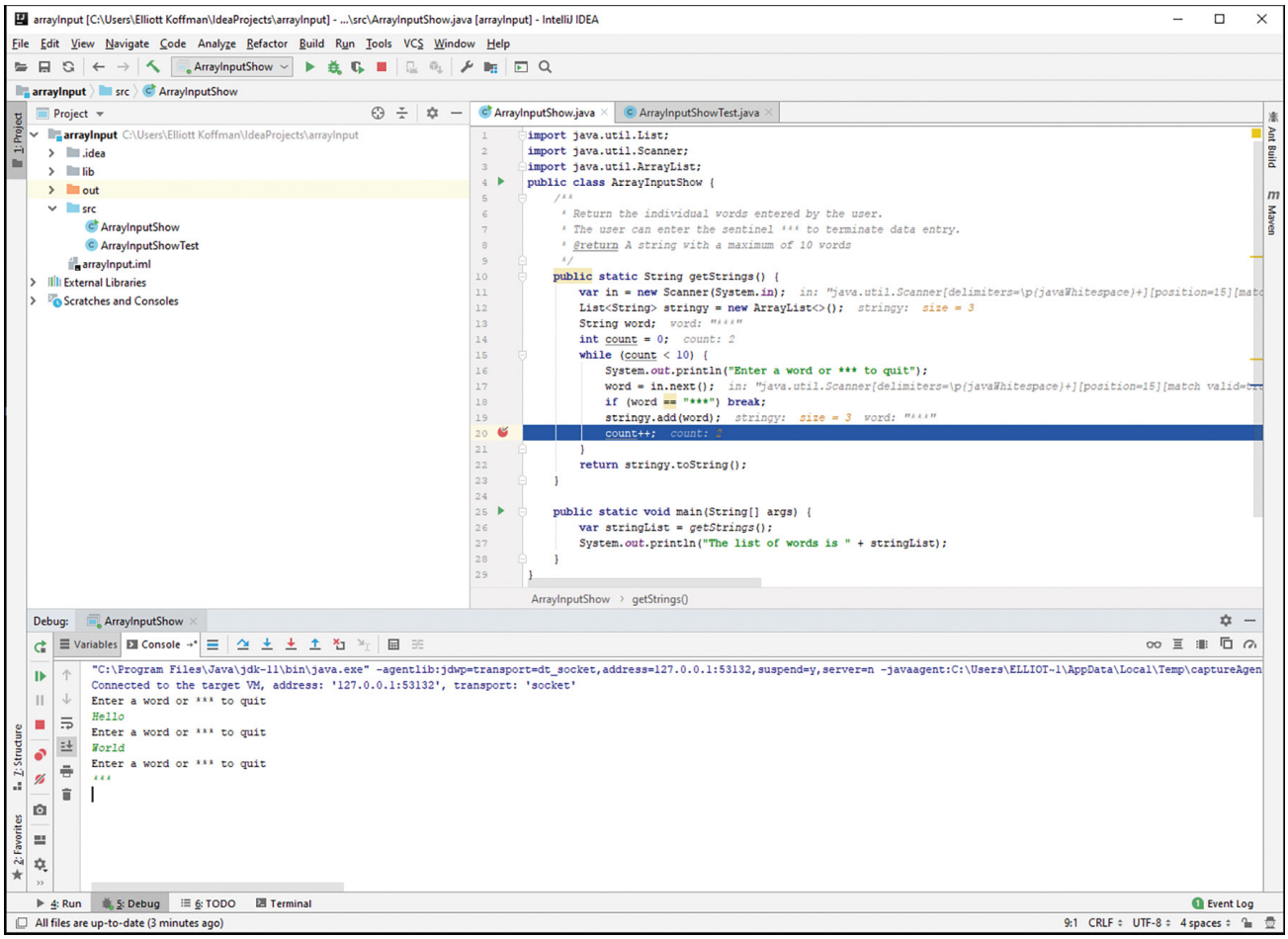
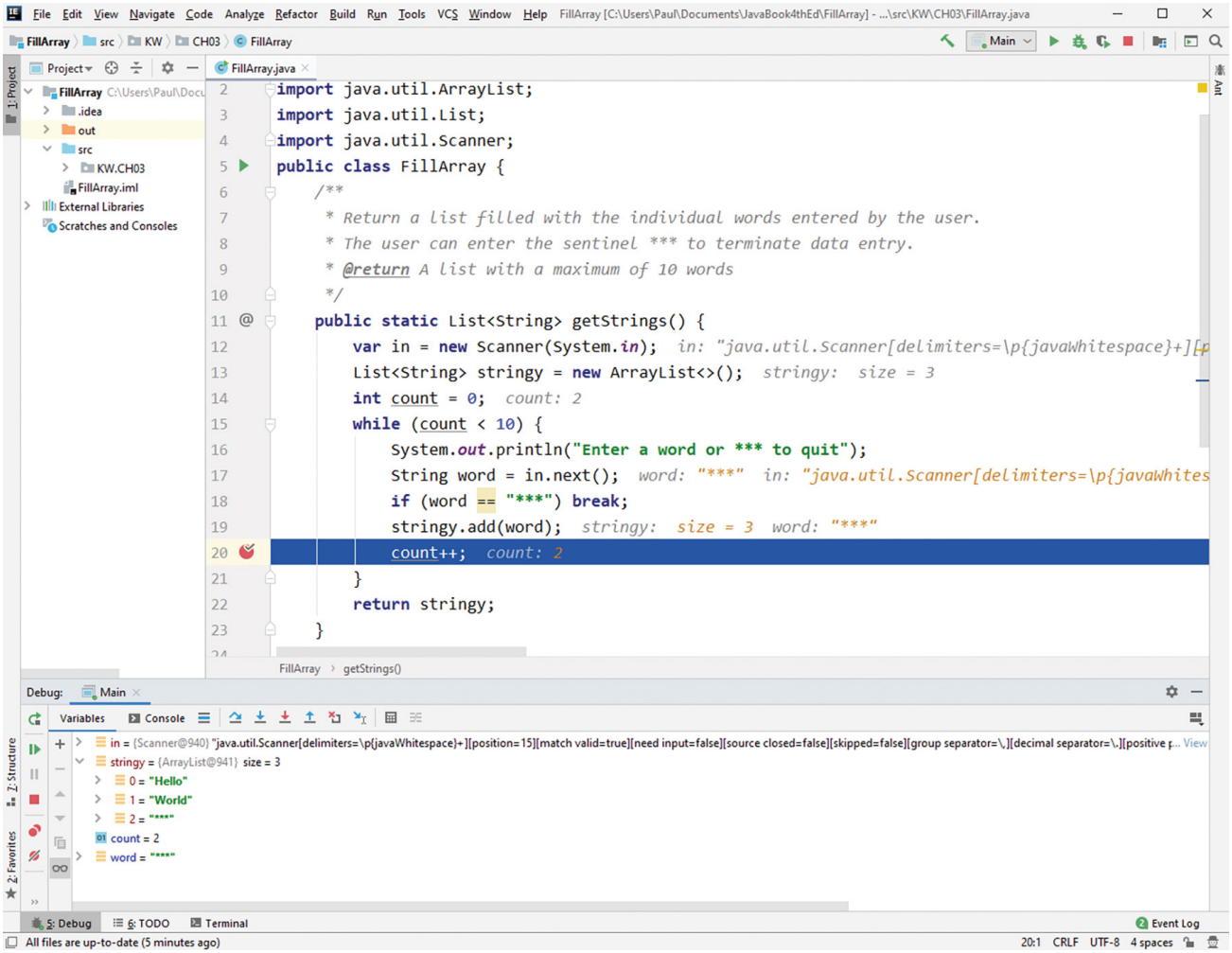


FIGURE 3.3 Edit Window (top right) and Variables Window (bottom) in IntelliJ

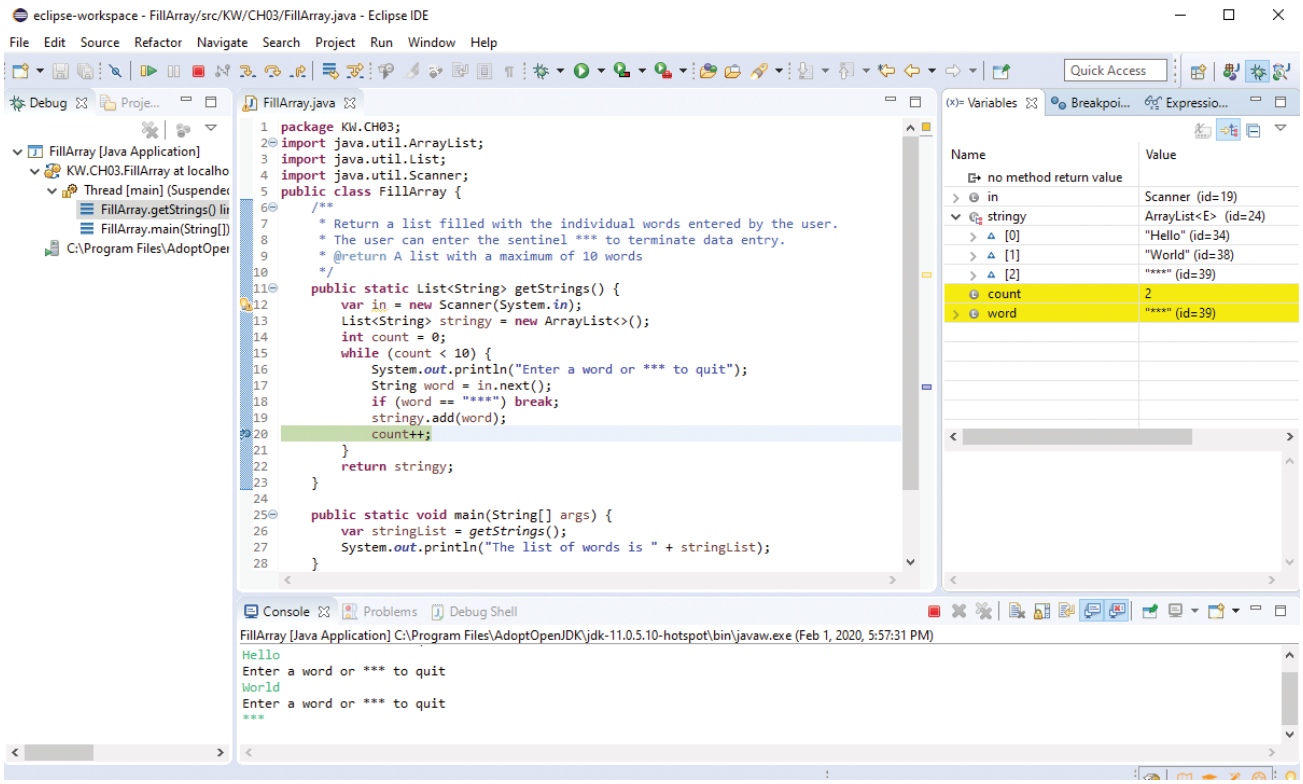


In IntelliJ (see Figure 3.3) you can open the Variables window to view all local variables when the program is suspended. Although we expected the condition (`word == "***"`) to be true, it is false, so loop exit did not occur and `"***"` was stored in `stringy`.

If you want to speed up the debugging process you could select Resume program after a stop at a breakpoint. The program will resume execution and pause when the next breakpoint is reached. In IntelliJ, press F9 or click the icon (▶) in the left column of the Debug window. (*In Eclipse, press F8 or click on the icon (▶) or select Resume from the Run-debug menu.*) Each time the program pauses, you can inspect the method's variables.

In Eclipse, when the breakpoint is reached, a dialog appears that lets you select the Debug view. This shows the Variables window to the right of the Edit window (Figure 3.4).

FIGURE 3.4 Edit Window (middle) and Variables Window (right) in Eclipse



EXERCISES FOR SECTION 3.7

SELF-CHECK

1. The following method does not appear to be working properly. Explain where you would add diagnostic output statements to debug it, and give an example of each statement.

```
/** Finds the largest value in array elements x[start] through x[last].
    @param x array whose largest value is found
    @param start first subscript in range
    @param last last subscript in range
    @return the largest value of x[start] through x[last]
    pre: start <= last
 */
public int findMax(int[] x, int start, int last) {
    if (start > last)
        throw new IllegalArgumentException("Empty range");
    int maxSoFar = 0;
    for (int i = start; i < last; i++) {
        if (x[i] > maxSoFar)
            maxSoFar = i;
    }
    return maxSoFar;
}
```

2. Explain the difference between selecting `Step Into` and `Step Over` during debugging.
3. If there was not a breakpoint in method `getStrings`, you could still trace its execution by setting a breakpoint at the statement that calls it. When the program was suspended at the breakpoint, would you use `Step Over`, `Step Into`, or both in order to trace the method's execution? Explain your answer.
4. How would the execution of method `getStrings` change if the breakpoint were set at the loop heading instead of at the last statement in the loop body?

PROGRAMMING

1. After debugging, provide a corrected version of the method in Self-Check Exercise 1. Leave the debugging statements in, but execute them only when the global constant `TESTING` is `true`.
2. Write and test a driver program to test method `findMax` in Self-Check Exercise 1.

Chapter Review

- Program testing is done at several levels starting with the smallest testable piece of the program, called a unit. A unit is either a method or a class, depending on the complexity.
- Once units are individually tested, they can be tested together; this level is called integration testing.
- Once the whole program is put together, it is tested as a whole; this level is called system testing.
- Finally, the program is tested in an operational manner demonstrating its functionality; this is called acceptance testing.
- Black-box (also called closed-box) testing tests the item (unit or system) based on its functional requirements without using any knowledge of the internal structure.
- White-box (also called glass-box or open-box) testing tests the item using knowledge of its internal structure. One of the goals of white-box testing is to achieve test coverage. This can range from testing every statement at least once, to testing each branch condition (`if` statements, `switch` statements, and loops) to verify each possible path through the program.
- Test drivers and stubs are tools used in testing. A test driver exercises a method or class and drives the testing. A stub stands in for a method that the unit being tested calls. This can be used to provide test results, and it can be used to enable a call of that method to be tested when the method being called is not yet coded.
- The JUnit test framework is a software product that facilitates writing test cases, running the test cases, and reporting the results.
- Test-Driven Development is an approach for developing programs that has gained popularity among professional software developers. The approach is to write test cases one at a time and the make the changes to the program required to pass the test.

- Interactive programs can be tested by using the `ByteArrayInputStream` to provide known input and the `ByteArrayOutputStream` to verify output.
- We described the debugging process and showed an example of how a debugger can be used to obtain information about a program's state.

Java API Classes Introduced in This Chapter

```
ByteArrayInputStream
ByteArrayOutputStream
```

User-Defined Interfaces and Classes in This Chapter

```
class ArraySearch
class MyInput
```

Quick-Check Exercises

1. _____ testing requires the use of test data that exercises each statement in a module.
2. _____ testing focuses on testing the functional characteristics of a module.
3. _____ determines whether a program has an error: _____ determines the _____ of the error and helps you _____ it.
4. A method's _____ and _____ serve as a _____ between a method caller and the method programmer—if a caller satisfies the _____, the method result should satisfy the _____.
5. Explain how the old adage "We learn from our mistakes" applies to test-driven development.

Review Questions

1. Indicate in which state of testing (unit, integration, system) each of the following kinds of errors should be detected:
   a. An array index is out of bounds.
   b. A `FileNotFoundException` is thrown.
   c. An incorrect value of withholding tax is being computed under some circumstances.
2. Describe the differences between stubs and drivers.
3. What is refactoring? How is it used in test-driven development?
4. Write a list of tests to verify the method described below.

```
/** Finds the largest value in array elements x[start] through x[last].
    @param x array whose largest value is found
    @param start first subscript in range
    @param last last subscript in range
    @return the largest value of x[start] through x[last]
    pre: start <= last
 */
```

5. Use the list of tests for Review Question 4, to develop a JUnit test class.
6. Use test-driven development to code a method for finding the smallest value in an array.

Programming Projects

1. Design and code a JUnit test class for a method that finds the smallest element in an array.
2. Develop the method in project 1 using test-driven development.
3. Design and code the JUnit test class for class `SinCos` (see Listing 3.4) that computes the sine and cosine functions in a specialized manner. This class is going to be part of an embedded system running on a processor that does not support floating-point arithmetic or the Java `Math` class. Your job

is to test the methods `sin` and `cos`; you are to assume that the methods `sin0to90` and `sin45to90` have already been tested.

You need to design a set of test data that will exercise each of the `if` statements. To do this, look at the boundary conditions and pick values that are

- Exactly on the boundary
- Close to the boundary
- Between boundaries

LISTING 3.4

SinCos.java

```
/** This class computes the sine and cosine of an angle expressed in
    degrees. The result will be an integer representing the sine or
    cosine as ten-thousandths.
 */
public class SinCos {
    /** Compute the sine of an angle in degrees.
        @param x The angle in degrees
        @return the sine of x
     */
    public static int sin(int x) {
        if (x < 0) {
            x = -x;
        }
        x = x % 360;
        if (0 <= x && x <= 45) {
            return sin0to45(x);
        } else if (45 <= x && x <= 90) {
            return sin45to90(x);
        } else if (90 <= x && x <= 180) {
            return sin(180 - x);
        } else {
            return -sin(x - 180);
        }
    }

    /** Compute the cosine of an angle in degrees
        @param x The angle in degrees
        @return the cosine of x
     */
    public static int cos(int x) {
        return sin(x + 90);
    }

    /** Compute the sine of an angle in degrees
        between 0 and 45
        @param x The angle
        @return the sine of x
        pre: 0 <= x < 45
     */
    private static int sin0to45(int x) {
        // In a realistic program this method would
        // use a polynomial approximation that was
        // optimized for the input range
        // Insert code to compute sin(x) for x between 0 and 45 degrees
    }
```

```
    /** Compute the sine of an angle in degrees between 45 and 90.
        @param x - The angle
        @return the sine of x
        pre: 45 <= x <= 90
     */
    private static int sin45to90(int x) {
        // In a realistic program this method would
        // use a polynomial approximation that was
        // optimized for the input range
        // Insert code to compute sin(x) for x between 45 and 90 degrees
    }
}
```

Answers to Quick-Check Exercises

1. *White-box* testing requires the use of test data that exercises each statement in a module.
2. *Black-box* testing focuses on testing the functional characteristics of a module.
3. *Testing* determines whether a program has an error: *debugging* determines the *cause* of the error and helps you *correct* it.
4. A method's *precondition* and *postcondition* serve as a *contract* between a method caller and the method programmer—if a caller satisfies the *precondition* the method result should satisfy the *postcondition*.
5. In test-driven development, failing a test informs the method coder that the method developed so far needs to be modified. By repeated failure and adaptation of the code to pass the test just failed while still passing the rest of the tests in the test class, the method coder develops a correct version of the desired method.

Chapter 4

Stacks, Queues, and Deques

Chapter Objectives

- To learn about stack data type and how to use its four methods: `push`, `pop`, `peek`, and `empty`
- To understand how Java implements a stack
- To learn how to implement a stack using an underlying array or a linked list
- To see how to use a stack to perform various applications, including finding palindromes and evaluating arithmetic expressions
- To learn how to represent a waiting line (queue) and how to use the methods in the `Queue` interface for insertion (`offer` and `add`), for removal (`remove` and `poll`), and for accessing the element at the front (`peek` and `element`)
- To understand how to implement the `Queue` interface using a single-linked list, a circular array, and a double-linked list
- To become familiar with the `Deque` interface and how to use its methods to insert and remove items from both ends of a deque

In this chapter we study two abstract data types, the stack and queue, that are widely used. A stack is a LIFO (last-in, first-out) list because the last element pushed onto a stack will be the first element to be popped off. A queue, on the other hand, is a FIFO (first-in, first-out) list because the first element inserted in the queue will be the first element removed.

Stacks and queues are more restrictive than the list data type that we studied in Chapter 2. A client can access any element in a list and can insert elements at any location. However, a client can access only a single element in a stack or queue: the one that was most recently inserted in the stack or the oldest one in the queue. This may seem like a serious restriction that would make them not very useful, but it turns out that stacks and queues are actually two of the most commonly used data structures in computer science. For example, during program execution, a stack is used to store information about the parameters and return points for all the methods that are currently executing (you will see how this is done in Chapter 5, "Recursion"). Compilers also use stacks to store information while evaluating expressions.

Operating systems also make extensive use of queues. One application of queues is to manage process execution. In a modern operating system, several processes may be executing

at the same time and, therefore, may request the use of the same computer resource (for example, the CPU or a printer). The operating system stores these requests in a queue and then removes them so that the process that is waiting the longest will be the next one to get access to the desired resource.

We will discuss several applications of stacks and queues. We will also show how to implement them using both arrays and linked lists.

Stacks, Queues, and Deques

4.1 Stack Abstract Data Type

In a cafeteria you can see stacks of dishes placed in spring-loaded containers. Usually, several dishes are visible above the top of the container, and the rest are inside the container. You can access only the dish that is on top of the stack. If you want to place more dishes on the stack, you can place the dishes on top of those that are already there. The spring inside the stack container compresses under the weight of the additional dishes, adjusting the height of the stack so that only the top few dishes are always visible.

FIGURE 4.1
A Pez Dispenser

Another physical example of a stack is a Pez® dispenser (see Figure 4.1). A Pez dispenser is a toy that contains candies. There is also a spring inside the dispenser. The top of the dispenser is a character's head. When you open the dispenser, a single candy *pops* out. You can only extract one candy at a time. If you want to eat more than one candy, you have to open the dispenser multiple times.

In programming, a stack is a data structure with the property that only the top element of the stack is accessible. In a stack, the top element is the data value that was most recently stored in the stack. Sometimes this storage policy is known as last-in, first-out, or *LIFO*.

Next, we specify some of the operations that we might wish to perform on a stack.

Specification of the Stack Abstract Data Type

Because only the top element of a stack is visible, a stack performs just a few operations. We need to be able to inspect the top element (method `peek`), retrieve the top element (method `pop`), push a new element onto the stack (method `push`), and test for an empty stack (method

TABLE 4.1
Specification of `StackInt<E>`

Methods	Behavior
`boolean isEmpty()`	Returns `true` if the stack is empty; otherwise, returns `false`
`E peek()`	Returns the object at the top of the stack without removing it
`E pop()`	Returns the object at the top of the stack and removes it
`E push(E obj)`	Pushes an item onto the top of the stack and returns the item pushed

`isEmpty`). Table 4.1 shows a specification for the Stack ADT that specifies the stack operations. We will write this as interface `StackInt<E>`.

Listing 4.1 shows the interface `StackInt<E>`, which declares the methods in the Stack ADT.

LISTING 4.1
StackInt.java

```
/** A Stack is a data structure in which objects are inserted into
    and removed from the same end i.e., Last-In, First-Out).
    @param <E> The element type
 */
public interface StackInt<E> {

    /** Pushes an item onto the top of the stack and returns
        the item pushed.
        @param obj The object to be inserted
        @return The object inserted
     */
    E push(E obj);

    /** Returns the object at the top of the stack
        without removing it.
        post: The stack remains unchanged.
        @return The object at the top of the stack
        @throws NoSuchElementException if stack is empty
     */
    E peek();

    /** Returns the object at the top of the stack and removes it.
          post: The stack is one item smaller.
          @return The object at the top of the stack
          @throws NoSuchElementException if stack is empty
     */
    E pop();

    /** Returns true if the stack is empty; otherwise,
        returns false.
        @return true if empty; otherwise, return false
     */
    boolean isEmpty();
}
```

EXAMPLE 4.1 A stack `names` (type `Stack<String>`) contains five strings as shown in Figure 4.2(a). The name `"Rich"` was placed on the stack before the other four names; `"Jonathan"` was the last element placed on the stack.

For stack `names` in Figure 4.2(a), the value of `names.isEmpty()` is `false`. The statement

```
String last = names.peek();
```

stores `"Jonathan"` in `last` without changing `names`. The statement

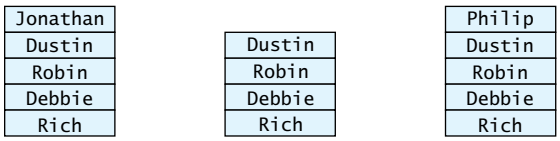
```
String temp = names.pop();
```

removes `"Jonathan"` from `names` and stores a reference to it in `temp`. The stack `names` now contains four elements and is shown in Figure 4.2(b). The statement

```
names.push("Philip");
```

pushes `"Philip"` onto the stack; the stack `names` now contains five elements and is shown in Figure 4.2(c).

FIGURE 4.2
Stack `Names`



(a) (b) (c)

EXERCISES FOR SECTION 4.1

SELF-CHECK

1. Assume that the stack `names` is defined as in Figure 4.2(c) and perform the following sequence of operations. Indicate the result of each operation and show the new stack if it is changed.

```
names.push("Jane");
String top = names.peek();
names.push("Joseph");
String nextTop = names.pop();
```

2. For the stack `names` in Figure 4.2 (c), what is the effect of the following:

```
while (!names.isEmpty()) {
    System.out.println(names.pop());
}
```

3. What would be the effect of using `peek` instead of `pop` in Question 2?

PROGRAMMING

1. Write a main function that creates three stacks of `Integer` objects. Store the numbers –1, 15, 23, 44, 4, 99 in the first two stacks. The top of each stack should store 99.
2. Write a loop to get each number from the first stack and store it into the third stack.
3. Write a second loop to remove a value from the second and third stacks and display each pair of values on a separate output line. Continue until the stacks are empty. Show the output.

4.2 Stack Applications

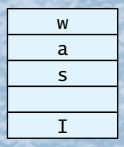
In this section we will study a client program that uses a stack, a palindrome finder. The `java.util.Stack` class is part of the original Java API but is not recommended for new applications. Instead, the Java designers recommend that we use the `java.util.Deque` interface and the `java.util.ArrayDeque` class to provide the methods listed in Table 4.1. The `Deque` interface specifies the methods in our interface `StackInt` (see Table 4.1) and also those needed for a queue. We discuss the `Deque` interface and class `ArrayDeque` in more detail in Section 4.8.

CASE STUDY Finding Palindromes

Problem A palindrome is a string that reads the same in either direction: left to right or right to left. For example, "kayak" is a palindrome, as is "I saw I was I." A well-known palindrome regarding Napoleon Bonaparte is "Able was I ere I saw Elba" (the island where he was sent in exile). We would like a program that reads a string and determines whether it is a palindrome.

Analysis This problem can be solved in many different ways. For example, you could set up a loop in which you compare the characters at each end of a string as you work toward the middle. If any pair of characters is different, the string can't be a palindrome. Another approach would be to scan a string backward (from right to left) and append each character to the end of a new string, which would become the reverse of the original string. Then you can see whether the strings are the same. The approach we will study here uses a stack to assist in forming the reverse of a string. It is not the most efficient way to solve the problem, but it makes good use of a stack.

If we scan the input string from left to right and push each character in the input string onto a stack, we can then form the reverse of the string by popping the characters and joining them together as they come off the stack. For example, the stack at left contains the characters in the input string `"I saw"`.



If we pop them off and join them together, we will get `"w" + "a" + "s" + " " + "I"`, or the string `"was I"`. When the stack is empty, we can compare the string we formed with the original and see if they are the same. If they are , the original string would be a palindrome. Because `char` is a primitive type, each character must be wrapped in a `Character` object before it can be pushed onto the stack.

Data Requirements

PROBLEM INPUTS

An input string to be tested

PROBLEM OUTPUTS

A message indicating whether the string is a palindrome

TABLE 4.2
Class `PalindromeFinder`

Methods	Behavior
`private static Deque<Character> fillStack(String inputString)`	Returns a stack that is filled with the characters in `inputString`
`private String buildReverse(String inputString)`	Calls `fillStack` to fill the stack based on `inputString` and returns a new string formed by popping each character from this stack and joining the characters. Empties the stack
`public boolean isPalindrome(String inputString)`	Returns `true` if `inputString` and the string built by `buildReverse` have the same contents, except for case. Otherwise, returns `false`

Design

We can define a class called `PalindromeFinder` (Table 4.2) with three static methods: `fillStack` pushes all characters from the input string onto a stack, `buildReverse` builds a new string by popping the characters off the stack and joining them, and `isPalindrome` compares the input string and new string to see whether they are palindromes. Method `isPalindrome` is the only public method and is called with the string to be tested as its argument.

Implementation

Listing 4.2 shows the class. Method `isPalindrome` calls `buildReverse`, which calls `fillStack` to build the stack (an `ArrayDeque`). In `fillStack`, the statement

```
charStack.push(inputString.charAt(i));
```

autoboxes a character and pushes it onto the stack.

In method `buildReverse`, the loop

```
while (!charStack.isEmpty()) {
    // Remove top item from stack and append it to result.
    result.append(charStack.pop());
}
```

pops each object off the stack and appends it to the result string.

Method `isPalindrome` uses the `String` method `equalsIgnoreCase` to compare the original `string` with its reverse.

```
return inputString.equalsIgnoreCase(buildReverse(inputString));
```

LISTING 4.2
PalindromeFinder.java

```
import java.util.*;

/** Class with methods to check whether a string is a palindrome. */
public class PalindromeFinder {

    /** Fills a stack of characters from an input string.
     *  @param inputString the string to be checked
     *  @return a stack of the characters in inputString
     */
    private static Deque<Character> fillStack(String inputString) {
        Deque<Character> charStack = new ArrayDeque<>();
```

```
            for (int i = 0; i < inputString.1ength(); i++) {
                charStack.push(inputString.charAt(i));
            }
            return charStack;
        }

        /**
         *  Builds the reverse of a string by calling fillStack
         *  to push its characters onto a stack and then popping them
         *  and appending them to a new string.
         *  post: The stack is empty.
         *  @return The string containing the characters in the stack
         */
        private static String buildReverse(String inputString) {
            Deque<Character> charStack = fillStack(inputString);
            StringBuilder result = new StringBuilder();
            while (!charStack.isEmpty()) {
                // Remove top item from stack and append it to result.
                result.append(charStack.pop());
            }
            return result.toString();
        }

        /** Calls buildReverse and compares its result to inputString
         *  @param inputString the string to be checked
         *  @return true if inputString is a palindrome, false if not
         */
        public static boolean isPalindrome(String inputString) {
            return inputString.equalsIgnoreCase(buildReverse(input String));
        }
    }
```

Testing

To test this class, you should run it with several different strings, including both palindromes and nonpalindromes, as follows:

- A single character (always a palindrome)
- Multiple characters in one word
- Multiple words
- Different cases
- Even-length strings
- Odd-length strings
- An empty string (considered a palindrome)

Listing 4.3 is a `JUnit` test class for the `PalindromeFinder` class.

LISTING 4.3
`PalindromeFinderTest`

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

/**
 *  Test of the PalindromeFinder
 *  @author Koffman & Wolfgang
 */
```

```
public class PalindromeFinderTest {
    public PalindromeFinderTest() {
    }

    @Test
    public void singleCharacterIsAIwaysAPalindrome() {
        assertTrue(PalindromeFinder.isPalindrome("x"));
    }

    @Test
    public void aSingleWordPalindrome() {
        assertTrue(PalindromeFinder.isPalindrome("kayak"));
    }

    @Test
    public void aSingleWordNonPalindrome() {
        assertFalse(PalindromeFinder.isPalindrome("foobar"));
    }

    @Test
    public void multipleWordsSameCase() {
        assertTrue(PalindromeFinder.isPalindrome("I saw I was I"));
    }

    @Test
    public void multipleWordsDifferentCase() {
        assertTrue(PalindromeFinder.isPalindrome
                   ("Able was I ere I saw Elba"));
    }

    @Test
    public void anEmptyStringIsAPalindrome() {
        assertTrue(PalindromeFinder.isPalindrome(""));
    }

    @Test
    public void anEvenLengthStringPalindrome() {
        assertTrue(PalindromeFinder.isPalindrome("foooof"));
    }
}
```

PITFALL

Attempting to Pop an Empty Stack

If you attempt to pop an empty stack, your program will throw a `NoSuchElementException`. You can guard against this error by testing for a nonempty stack before popping the stack. Alternatively, you can catch the error if it occurs and handle it.

Stack Application on Textbook Website

There is an additional case study on the textbook website that uses a stack to check that the parentheses in an expression are balanced. This can be found at the URL specified in the preface.

EXERCISES FOR SECTION 4.2

SELF-CHECK

1. The result returned by the palindrome finder depends on all characters in a string, including spaces and punctuation. Discuss how you would modify the palindrome finder so that only the letters and digits in the input string are used to determine whether the input string is a palindrome. You should ignore any other characters.

PROGRAMMING

1. Write a method that reads a line and reverses the words in the line (not the characters) using a stack. For example, given the following input:

   ```
   The quick brown fox jumps over the lazy dog
   ```

   you should get the following output:

   ```
   dog lazy the over jumps fox brown quick The
   ```

2. Three different approaches to finding palindromes are discussed in the Analysis section of that case study. Code the first approach.
3. Code the second approach to find palindromes.

4.3 Implementing a Stack

This section discusses how to implement our stack interface (`StackInt`). We will show how to do this using class `ArrayList` and also using class `LinkedList`.

Implementing a Stack with an `ArrayList` Component

You may have recognized that a `stack` is very similar to an `ArrayList`. In fact, in the Java Collections framework, the class `Stack` extends class `Vector`, which is the historical predecessor of `ArrayList`. Just as they suggest using interface `Deque` instead of the `Stack` class in new applications, the Java designers recommend using class `ArrayList` instead of class `Vector`.

Next we show how to write a class, which we will call `ListStack`, that has an `ArrayList` component. We will call this component `theData`, and it will contain the stack data.

We code the `ListStack<E>.push` method as:

```
public E push(E obj) {
    theData.add(obj);
    return obj;
}
```

The `ListStack` class is said to be an *adapter class* because it adapts the methods available in another class (`a List`) to the interface its clients expect by giving different names to essentially the same operations (e.g., `push` instead of `add`). This is an example of *method delegation*.

Listing 4.4 shows the `ListStack` class. Note that the statements that manipulate the stack explicitly refer to data field `theData`. For example, in `ListStack` push we use the statement

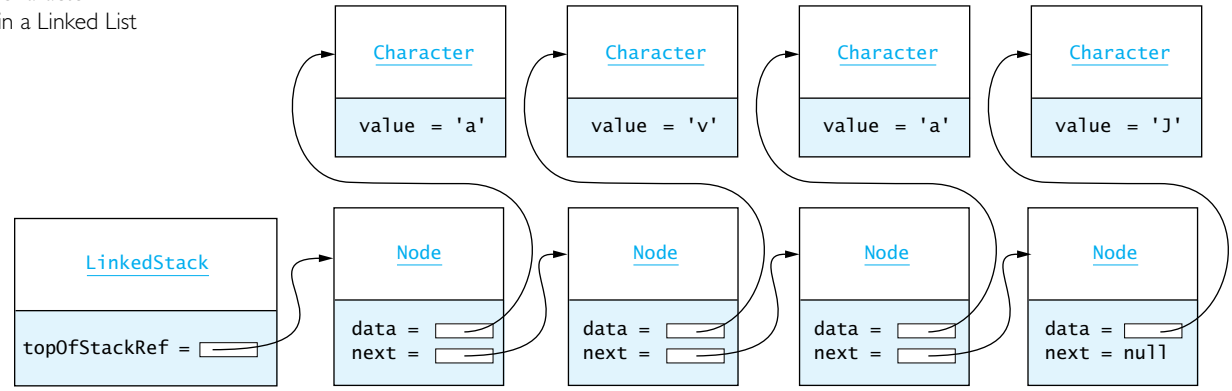
```
theData.add(obj);
```

to push `obj` onto the stack as the new last element of `ArrayList theData`.

LISTING 4.4
ListStack.java

```
import java.util.*;

/** Class ListStack<E> implements the interface StackInt<E> as
    an adapter to the List.
    @param <E> The type of elements in the stack.
 */
 public class ListStack<E> implements StackInt<E> {

    /** The List containing the data */
    private List<E> theData;

    /** Construct an empty stack using an ArrayList as the container. */
    public ListStack() {
        theData = new ArrayList<>();
    }

    /** Push an object onto the stack.
        post: The object is at the top of the stack.
        @param obj The object to be pushed
        @return The object pushed
     */
    @Override
    public E push(E obj) {
        theData.add(obj);
        return obj;
    }

    /** Peek at the top object on the stack.
        @return The top object on the stack
        @throws NoSuchElementException if the stack is empty
     */
    @Override
    public E peek() {
        if (isEmpty()) {
            throw new NoSuchElementException();
        }
        return theData.get(theData.size() - 1);
    }

    /** Pop the top object off the stack.
        post: The object at the top of the stack is removed.
        @return The top object, which is removed
        @throws NoSuchElementException if the stack is empty
     */
    @Override
    public E pop() {
        if (isEmpty()) {
            throw new NoSuchElementException();
        }
        return theData.remove(theData.size() - 1);
    }

    /** See whether the stack is empty.
        @return true if the stack is empty
     */
    @Override
    public boolean isEmpty() {
        return theData.isEmpty();
    }
}
```

FIGURE 4.3
Stack of Character Objects in a Linked List



Implementing a Stack as a Linked Data Structure

We can also implement a stack using a single-linked list of nodes. We show the stack containing the characters in "Java" in Figure 4.3, with the last character in the string stored in the node at the top of the stack. Class `LinkedStack<E>` contains a collection of `Node<E>` objects (see Section 2.5). Recall that inner class `Node<E>` has attributes `data` (type `E`) and `next` (type `Node<E>`).

Reference variable `topOfStackRef` (type `Node<E>`) references the last element placed on the stack. Because it is easier to insert and delete from the head of a linked list, we will have `topOfStackRef` reference the node at the head of the list.

Method `push` inserts a node at the head of the list. The statement

```
topOfStackRef = new Node<>(obj, topOfStackRef);
```

resets `topOfStackRef` to reference the new node; `topOfStackRef.next` references the old top of the stack. When the stack is empty, `topOfStackRef` is `null`, so the attribute `next` for the first object pushed onto the stack (the item at the bottom) will be `null`.

Method `peek` will be very similar to the `LinkedList` method `getFirst`. Method `isEmpty` tests for a value of `topOfStackRef` equal to `null`. Method `pop` simply resets `topOfStackRef` to the value stored in the `next` field of the list head and returns the old `topOfStackRef` data. Listing 4.5 shows class `LinkedStack`.

LISTING 4.5
Class `LinkedStack`

```
import java.util.NoSuchElementException;

/** Class to implement interface StackInt<E> as a linked list. */
public class LinkedStack<E> implements StackInt<E> {
    // Insert inner class Node<E> here. (See Listing 2.1)

    // Data Fields
    /** The reference to the first stack node. */
    private Node<E> topOfStackRef = null;

    /** Insert a new item on top of the stack.
        post: The new item is the top item on the stack.
```

```
            All other items are one position lower.
        @param obj The item to be inserted
        @return The item that was inserted
     */
    @Override
    public E push(E obj) {
        topOfStackRef = new Node<>(obj, topOfStackRef);
        return obj;
    }

    /** Remove and return the top item on the stack.
        pre: The stack is not empty.
        post: The top item on the stack has been
              removed and the stack is one item smaller.
        @return The top item on the stack
        @throws NoSuchElementException if the stack is empty
     */
    @Override
    public E pop() {
        if (isEmpty()) {
            throw new NoSuchElementException();
        }
        else {
            E result = topOfStackRef.data;
            topOfStackRef = topOfStackRef.next;
            return result;
        }
    }

    /** Return the top item on the stack.
        pre: The stack is not empty.
        post: The stack remains unchanged.
        @return The top item on the stack
        @throws NoSuchElementException if the stack is empty
     */
    @Override
    public E peek() {
        if (isEmpty()) {
            throw new NoSuchElementException ();
        }
        else {
            return topOfStackRef.data;
        }
    }

    /** See whether the stack is empty.
        @return true if the stack is empty
     */
    @Override
    public boolean isEmpty() {
          return topOfStackRef == null;
     }
}
```

Comparison of Stack Implementations

The easiest approach to implementing a stack in Java would be to give it a `List` component for storing the data. Since all insertions and deletions are at one end, the stack operations would all be O(1) operations. You could use an object of any class that implements the `List` interface to store the stack data, but the `ArrayList` is the simplest.

Finally, you could also use your own linked data structure. This has the advantage of using exactly as much storage as is needed for the stack. However, you would also need to allocate storage for the links. Because all insertions and deletions are at one end, the flexibility provided by a linked data structure is not utilized. All stack operations using a linked data structure would also be O(1).

EXERCISES FOR SECTION 4.3

SELF-CHECK

1. For the implementation of stack `s` using an `ArrayList` as the underlying data structure, show how the underlying data structure changes after each statement below executes. Assume the characters in `"Happy"` are already stored on the stack (`H` pushed on first).

```
s.push('*');
s.push('i');
char ch1 = s.pop();
s.pop();
s.push(' ');
char ch2 = s.peek();
```

2. How do your answers to Question 1 change if the initial capacity is 4 instead of 7?
3. For the implementation of stack `s` using a linked list of nodes as the underlying data structure (see Figure 4.3), show how the underlying data structure changes after each of the following statements executes. Assume the characters in `"Happy"` are already stored on the stack (`H` pushed on first).

```
s.push('i');
s.push('s');
char ch1 = s.pop();
s.pop();
s.push(' ');
char ch2 = s.peek();
```

PROGRAMMING

1. Write a method `size` for class `LinkedStack<E>` that returns the number of elements currently on a `LinkedStack<E>`.
2. Write method `size` for the `ArrayList` implementation.

4.4 Additional Stack Applications

In this section we consider two case studies that relate to evaluating arithmetic expressions. The first problem is slightly easier, and it involves evaluating expressions that are in postfix form. The second problem discusses how to convert from *infix notation* (common mathematics notation) to postfix form.

Normally we write expressions using infix notation, in which binary operators (`*`, `+`, etc.) are inserted between their operands. Infix expressions present no special problem to humans because we can easily scan left and right to find the operands of a particular operator.

TABLE 4.3
Postfix Expressions

Postfix Expression	Infix Expression	Value
4 7 *	4 * 7	28
4 7 2 + *	4 * (7 + 2)	36
4 7 * 20 –	(4 * 7) – 20	8
3 4 7 * 2 / +	3 + ((4 * 7) / 2)	17

A calculator (or computer), however, normally scans an expression string in the order that it is input (left to right). Therefore, it is easier to evaluate an expression if the user types in the operands for each operator before typing the operator (*postfix notation*). Table 4.3 shows some examples of expressions in postfix and infix form. The braces under each postfix expression will help you visualize the operands for each operator.

The advantage of the postfix form is that there is no need to group subexpressions in parentheses or even to consider operator precedence. (We talk more about postfix form in the second case study in this section.) The next case study develops a program that evaluates a postfix expression.

CASE STUDY Evaluating Postfix Expressions

Problem Write a class that evaluates a postfix expression. The postfix expression will be a string containing digit characters and operator characters from the set +, –, *, /. The space character will be used as a delimiter between tokens (integers and operators).

Analysis In a postfix expression, the operands precede the operators. A stack is the perfect place to save the operands until the operator is scanned. When the operator is scanned, its operands can be popped off the stack (the last operand scanned, i.e., the right operand, will be popped first). Therefore, our program will push each integer operand onto the stack. When an operator is read, the top two operands are popped, the operation is performed on its operands, and the result is pushed back onto the stack. The final result should be the only value remaining on the stack when the end of the expression is reached.

Design We will write class `PostfixEvaluator` to evaluate postfix expressions. The class should define a method `eval`, which scans a postfix expression and processes each of its tokens, where a token is either an operand (an integer) or an operator. We also need a method `evalOp`, which evaluates each operator when it is scanned, and a method `isOperator`, which determines whether a character is an operator. Table 4.4 describes the class.

The algorithm for `eval` follows. The stack operators perform algorithm steps 1, 5, 7, 8, 10, and 11.

Table 4.5 shows the evaluation of the third expression in Table 4.3 using this algorithm. The arrow under the expression points to the character being processed and is shown in color; the stack diagram shows the stack after this character is processed.

TABLE 4.4
Class `PostfixEvaluator`

Method	Behavior
`public static int eval(String expression)`	Returns the value of expression
`private static int evalOp(char op, Deque<Integer> operandStack)`	Pops two operands and applies operator `op` to its operands, returning the result
`private static boolean isOperator(char ch)`	Returns `true` if `ch` is an operator symbol

TABLE 4.5
Evaluating a Postfix Expression

Expression	Action	Stack
4 7 * 20 – ↑	Push 4	4
4 7 * 20 – ↑	Push 7	7 4
4 7 * 20 – ↑	Pop 7 and 4 Evaluate 4 * 7 Push 28	28
4 7 * 20 – ↑	Push 20	20 28
4 7 * 20 – ↑	Pop 20 and 28 Evaluate 28 – 20 Push 8	8
4 7 20 – ↑	At end of expression Pop 8 Stack is empty Result is 8	

Algorithm for method eval

1. Create an empty stack of integers.
2. `while` there are more tokens
3. Get the next token.
4. `if` the first character of the token is a digit.
5. Push the integer onto the stack.
6. `else if` the token is an operator
7. Pop the right operand off the stack.
8. Pop the left operand off the stack.
9. Evaluate the operation.
10. Push the result onto the stack.
11. Pop the stack and return the result.

Implementation

Listing 4.6 shows the implementation of class `PostfixEvaluator`. There is an inner class that defines the exception `SyntaxErrorException`.

Method `eval` implements the algorithm shown in the design section. To simplify the extraction of tokens, we will assume that there are spaces between operators and operands. As explained in Appendix A.5, the method call

```
String[] tokens = expression.split("\\s+");
```

stores in array `tokens` the individual tokens (operands and operators) of string `expression` where the argument string `"\\s+"` specifies that the delimiter is one or more white-space characters. (We will remove the requirement for spaces between tokens and consider parentheses in the last case study of this section.)

The enhanced `for` statement

```
for (String nextToken : tokens) {
```

ensures that each of the strings in `tokens` is processed, and the `if` statement in the loop tests the first character of each token to determine its category (number or operator). Therefore, the body of method `eval` is enclosed within a `try-catch` sequence. A `NoSuchElementException`, thrown either as a result of a `pop` operation in `eval` or by a `pop` operation in a method called by `eval`, will be caught by the `catch` clause. In either case, a `SyntaxErrorException` is thrown.

Private method `isOperator` determines whether a character is an operator. When an operator is encountered, private method `evalOp` is called to evaluate it. This method pops the top two operands from the stack. The first item popped is the right-hand operand, and the second is the left-hand operand.

```
int rhs = operandStack.pop();
int lhs = operandStack.pop();
```

A `switch` statement is then used to select the appropriate expression to evaluate for the given operator. For example, the following `case` processes the addition operator and saves the sum of `lhs` and `rhs` in `result`.

```
case '+' : result = lhs + rhs; break;
```

LISTING 4.6

PostfixEvaluator.java

```
import java.util.*;

/** Class that can evaluate a postfix expression. */
public class PostfixEvaluator {

    // Nested Class
    /** Class to report a syntax error. */
    public static class SyntaxErrorException extends Exception {
        /** Construct a SyntaxErrorException with the specified message.
            @param message The message
         */
        SyntaxErrorException(String message) {
            super(message);
        }
    }

    // Constant
    /** A list of operators. */
    private static final String OPERATORS = "+-*/";
```

```
// Methods
/** Evaluates the current operation.
    This function pops the two operands off the operand stack and applies
    the operator.
    @param op A character representing the operator
    @param operandStack the current stack of operands
    @return The result of applying the operator
    @throws NoSuchElementException if pop is attempted on an empty stack
 */
private static int evalOp(char op, Deque<Integer> operandStack) {
    // Pop the two operands off the stack.
    int rhs = operandStack.pop();
    int lhs = operandStack.pop();
    int result = 0;
    // Evaluate the operator.
    switch (op) {
        case '+' : result = lhs + rhs;
                   break;
        case '-' : result = lhs - rhs;
                   break;
        case '/' : result = lhs / rhs;
                   break;
        case '*' : result = lhs * rhs;
                   break;
    }
    return result;
}

/** Determines whether a character is an operator.
    @param op The character to be tested
    @return true if the character is an operator
 */
private static boolean isOperator(char ch) {
    return OPERATORS.indexOf(ch) != -1;
}

/** Evaluates a postfix expression.
    @param expression The expression to be evaluated
    @return The value of the expression
    @throws SyntaxErrorException if a syntax error is detected
 */
public static int eval(String expression) throws SyntaxErrorException {
    // Create an empty stack.
    Deque<Integer> operandStack = new ArrayDeque<>();

    // Process each token.
    String[] tokens = expression.split("\\s+");
    try {
        for (String nextToken : tokens) {
            char firstChar = nextToken.charAt(0);
            // Does it start with a digit?
            if (Character.isDigit(firstChar)) {
                // Get the integer value.
                int value = Integer.parseInt(nextToken);
                // Push value onto operand stack.
                operandStack.push(value);
            } // Is it an operator?
```

```
                else if (isOperator(firstChar)) {
                    // Evaluate the operator.
                    int result = evalOp(firstChar, operandStack);
                    // Push result onto the operand stack.
                    operandStack.push(result);
                }
                else {
                    // Invalid character.
                    throw new SyntaxErrorException
                          ("Invalid character encountered: " + firstChar);
                }
            } // End for

            // No more tokens - pop result from operand stack.
            int answer = operandStack.pop();
            // Operand stack should be empty.
            if (operandStack.isEmpty()) {
                return answer;
            } else {
                // Indicate syntax error.
                throw new SyntaxErrorException
                      ("Syntax Error: Stack should be empty");
            }
        } catch (NoSuchElementException ex) {
            // Pop was attempted on an empty stack.
            throw new SyntaxErrorException("Syntax Error: Stack is empty");
        }
    }
}
```

PROGRAM STYLE

Creating Your Own Exception Class

The program would work just the same if we did not bother to declare the `SyntaxErrorException` class and just threw a new `Exception` object each time an error occurred. However, we feel that this approach gives the user a more meaningful description of the cause of an error. Also, if other errors are possible in a client of this class, any `SyntaxErrorException` can be caught and handled in a separate `catch` clause.

Testing You will need to write a `JUnit` test class for the `PostfixEvaluator` class. Each test case should pass an expression to `eval` and compare the expected and actual results. A white-box approach to testing would lead you to consider the following test cases. First, you want to exercise each path in the `evalOp` method by entering a simple expression that uses each operator. Then you need to exercise the paths through `eval` by trying different orderings and multiple occurrences of the operators. These tests exercise the normal cases, so you next need to test for possible syntax errors. You should consider the following cases: an operator without any operands, a single operand, an extra operand, an extra operator, a variable name, and finally an empty string.

CASE STUDY Converting from Infix to Postfix

We normally write expressions in infix notation. Therefore, one approach to evaluating expressions in infix notation is first to convert it to postfix and then to apply the evaluation technique just discussed. We will show in this case study how to accomplish this conversion using a stack. An infix expression can also be evaluated directly using two stacks. This is left as a programming project.

Problem

To complete the design of an expression evaluator, we need a set of methods that convert infix expressions to postfix form. We will assume that the expression will consist only of spaces, operands, and operators, where the space is a delimiter character between tokens. All operands that are identifiers begin with a letter or underscore character; all operands that are numbers begin with a digit. (Although we are allowing for identifiers, our postfix evaluator can't really handle them.)

Analysis

Table 4.3 showed the infix and postfix forms of four expressions. For each expression pair, the operands are in the same sequence; however, the placement of the operators changes in going from infix to postfix. For example, in converting

```
w - 5.1 / sum * 2
```

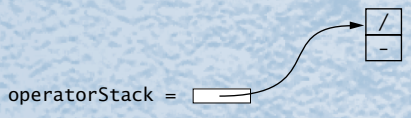
to its postfix form

```
w 5.1 sum / 2 * -
```

we see that the four operands (the tokens `w`, `5.1`, `sum`, `2`) retain their relative ordering from the infix expression, but the order of the operators is changed. The first operator in the infix expression, –, is the last operator in the postfix expression. Therefore, we can insert the operands in the output expression (`postfix`) as soon as they are scanned in the input expression (`infix`), but each operator should be inserted in the postfix string after its operands and in the order in which they should be evaluated, not the order in which they were scanned. For expressions without parentheses, there are two criteria that determine the order of operator evaluation:

- Operators are evaluated according to their *precedence* or rank. Higher precedence operators are evaluated before lower precedence operators. For example, `*`, `/`, and `%` (the *multiplicative* operators) are evaluated before `+`, `-`.
- Operators with the same precedence are evaluated in left-to-right order (left-associative rule).

If we temporarily store the operators on a stack, we can pop them whenever we need to and insert them in the postfix string in an order that indicates when they should be evaluated, rather than when they were scanned. For example, if we have the first two operators from the string `"w - 5.1 / sum * 2"` stored on a stack as follows,



the operator / (scanned second) must come off the stack and be placed in the postfix string before the operator – (scanned first). If we have the stack as just shown and the next operator is *, we need to pop the / off the stack and insert it in the postfix string before *, because the multiplicative operator scanned earlier (/) should be evaluated before the multiplicative operator (*) scanned later (the left-associative rule).

TABLE 4.6
Class `InfixToPostfix`

Data Field	Attribute
`private static final String OPERATORS`	The operators
`private static final int[] PRECEDENCE`	The precedence of the operators, matches their order in `OPERATORS`
`private Deque<Character> operatorStack`	Stack of operators
`private StringJoiner postfix`	The postfix string being formed
Method	**Behavior**
`public static String convert(String infix)`	Instantiates an instance of the `InfixToPostfix` class, calls `convertToPostfix` and then returns the result of calling `getPostifx`
`public void convertToPostfix(String infix)`	Extracts and processes each token in `infix` and stores the result in `StringJoiner postfix`
`private void processOperator(char op)`	Processes operator `op` by updating `operatorStack` and `postfix`
`private String getPostfix()`	Returns the result of `postfix.toString()`
`private static int precedence(char op)`	Returns the precedence of operator `op`
`private static boolean isOperator(char ch)`	Returns `true` if `ch` is an operator symbol

Design

Class `InfixToPostfix` contains methods needed for the conversion. The class should have a data field `operatorStack`, which stores the operators. It should also have a method `convert`, which does the initial processing of all tokens (operands and operators). Method `convert` needs to get each token and process it. Each token that is an operand should be appended to the postfix string. Method `processOperator` will process each operator token. Method `isOperator` determines whether a token is an operator, and method `precedence` returns the precedence of an operator. Table 4.6 describes class `InfixToPostfix`.

The algorithm for method `convert` follows. The `while` loop extracts and processes each token, calling `processOperator` to process each operator token. After all tokens are extracted from the infix string and processed, any operators remaining on the stack should be popped and appended to the postfix string. They are appended to the end because they have lower precedence than those operators inserted earlier.

Algorithm for Method convert

1. Initialize `postfix` to an empty `StringJoiner`.
2. Initialize the operator stack to an empty stack.
3. `while` there are more tokens in the infix string.
4. Get the next token.
5. `if` the next token is an operand.
6. Append it to postfix.
7. `else if` the next token is an operator.
8. Call `processOperator` to process the operator.
9. `else`

10. Indicate a syntax error.
11. Pop remaining operators off the operator stack and append them to `postfix`.

Method `processOperator`

The real decision making happens in method `processOperator`. By pushing operators onto the stack or popping them off the stack (and into the postfix string), this method controls the order in which the operators will be evaluated.

Each operator will eventually be pushed onto the stack. However, before doing this, `processOperator` compares the operator's precedence with that of the stacked operators, starting with the operator at the top of the stack. If the current operator has higher precedence than the operator at the top of the stack, it is pushed onto the stack immediately. This will ensure that none of the stacked operators can be inserted into the postfix string before it.

However, if the operator at the top of the stack has higher precedence than the current operator, it is popped off the stack and inserted in the postfix string, because it should be performed before the current operator, according to the precedence rule. Also, if the operator at the top of the stack has the same precedence as the current operator, it is popped off the stack and inserted into the postfix string, because it should be performed before the current operator, according to the left-associative rule. After an operator is popped off the stack, we repeat the process of comparing the precedence of the operator now at the top of the stack with the precedence of the current operator until the current operator is pushed onto the stack.

A special case is an empty operator stack. In this case, there are no stacked operators to compare with the new one, so we will simply push the current operator onto the stack. We use method `peek` to access the operator at the top of the stack without removing it.

Algorithm for Method `processOperator`

1. `if` the operator stack is empty
2. Push the current operator onto the stack.

`else`

3. Peek the operator stack and let `topOp` be the top operator.
4. `if` the precedence of the current operator is greater than the precedence of `topOp`
5. Push the current operator onto the stack.

`else`

6. `while` the stack is not empty and the precedence of the current operator is less than or equal to the precedence of `topOp`
7. Pop `topOp` off the stack and append it to `postfix`.
8. `if` the operator stack is not empty
9. Peek the operator stack and let `topOp` be the top operator.
10. Push the current operator onto the stack.

Table 4.7 traces the conversion of the infix expression `w - 5.1 / sum * 2` to the postfix expression `w 5.1 sum / 2 * -`. The final value of `postfix` shows that / is performed first (operands `5.1` and `sum`), * is performed next (`operands 5.1 / sum and 2`), and - is performed last.

Although the algorithm will correctly convert a well-formed expression and will detect some expressions with invalid syntax, it doesn't do all the syntax checking required. For

TABLE 4.7
Conversion of `w - 5.1 / sum * 2`

Next Token	Action	Effect on operatorStack	Effect on postfix
`w`	Append `w` to postfix		`w`
`-`	The stack is empty Push – onto the stack	`-`	`w`
`5.1`	Append 5.1 to postfix	`-`	`w 5.1`
`/`	`precedence(/) > precedence(-)`, Push / onto the stack	`/` `-`	`w 5.1`
`sum`	Append `sum` to `postfix`	`/` `-`	`w 5.1 sum`
`*`	`precedence(*)` equals `precedence(/)` Pop / off of stack and append to postfix	`-`	`w 5.1 sum /`
`*`	`precedence(*) > precedence(-)`, Push * onto the stack	`*` `-`	`w 5.1 sum /`
`2`	Append 2 to `postfix`	`*` `-`	`w 5.1 sum / 2`
`End of input`	Stack is not empty, Pop * off the stack and append to `postfix`	`-`	`w 5.1 sum / 2 *`
`End of input`	Stack is not empty, Pop – off the stack and append to `postfix`		`w 5.1 sum / 2 * -`

example, an expression with extra operands would not be detected. We discuss this further in the testing section.

Implementation

Listing 4.7 shows the `InfixToPostfix` class. A client would call method `convert`, passing it an infix string. Method `convert` creates object `infixToPostfix` and calls `convertToPostfix` to convert the argument string to postfix and return the conversion result.

The `convertToPostfix` method begins by initializing `postfix` and the `operatorStack`. The tokens are extracted using `String.split` and processed within a `try block`. The condition

```
(Character.isJavaIdentifierStart(firstChar)
 || Character.isDigit(firstChar))
```

tests the first character (`firstChar`) of the next token to see whether the next token is an operand (identifier or number). Method `isJavaIdentifierStart` returns `true` if the next token is an identifier; method `isDigit` returns `true` if the next token is a number (starts with a digit). If this condition is true, the token is appended to `postfix`. The next condition,

```
(isOperator(firstChar))
```

is true if `nextToken` is an operator. If so, method `processOperator` is called. If the next token is not an operand or an operator, the exception `SyntaxErrorException` is thrown.

Once the end of the expression is reached, the remaining operators are popped off the stack and appended to `postfix`.

Method `processOperator` uses private method `precedence` to determine the precedence of an operator (2 for *, /; 1 for +, -.). If the stack is empty or the condition

```
(precedence(op) > precedence(topOp))
```

is true, the current operator, `op`, is pushed onto the stack. Otherwise, the `while` loop executes, popping all operators off the stack that have the same or greater precedence than `op` and appending them to the `postfix` string (a `StringJoiner`).

```
while (!operatorStack.isEmpty()
       && precedence(op) <= precedence(topOp)) {
    operatorStack.pop();
    postfix.add(
    Character.toString(topOp))
}
```

After loop exit, the statement

```
operatorStack.push(op);
```

pushes the current operator onto the stack.

In method `precedence`, the statement

```
return PRECEDENCE[OPERATORS.indexOf(op)];
```

returns the element of `int[]` array `PRECEDENCE` selected by the method call `OPERATORS.indexOf(op)`. The precedence value returned will be 1 or 2.

LISTING 4.7

InfixToPostfix.java

```
import java.util.*;

/** Translates an infix expression to a postfix expression. */
public class InfixToPostfix {

    // Insert nested class SyntaxErrorException. See Listing 4.6

    // Data Fields
    /** The operator stack */
    /private final Deque<Character> operatorStack = new ArrayDeque<>();
    /** The operators */
    private static final String OPERATORS = "+-*/";
    /** The precedence of the operators matches order in OPERATORS. */
    private static final int[] PRECEDENCE = {1, 1, 2, 2};
    /** The postfix string */
    private final StringJoiner postfix = new StringJoiner<>(" ");

    /** Convert a string from infix to postfix. Public convert is called
     * by a client - Calls private method convertToPostfix to do the conversion.
     * @param infix The infix expression
     * @throws SyntaxErrorException
     * @return the equivalent postfix expression.
     */
    public static String convert(String infix)
           throws SyntaxErrorException {
        InfixToPostfix = new InfixToPostfix();
        infixToPostfix.convertToPostfix(infix);
        return infixToPostfix.getPostfix();
    }
```

```
/** Return the final postfix string. */
private String getPostfix() {return postfix.toString();}

/** Convert a string from infix to postfix. Public convert is called
 *  by a client - Calls private method convertToPostfix to do the conversion.
 *  Uses a stack to convert an infix expression to postfix
 *  pre: operator stack is empty
 *  post: postFix contains postfix expression and stack is empty
 *  @param infix the string to convert to postfix
 *  @throws SyntaxErrorException if argument is invalid
 */
private void convertToPostfix(String infix) throws SyntaxErrorException {
    String[] tokens = infix.split("\\s+");
    try {
        // Process each token in the infix string.
        for (String nextToken : tokens) {
            char firstChar = nextToken.charAt(0);
            // Is it an operand?
            if (Character.isJavaIdentifierStart(firstChar)
                || Character.isDigit(firstChar)) {
                postfix.add(nextToken);
            } // Is it an operator?
            else if(isOperator(firstChar)) {
                processOperator(firstChar);
            }
            else {
                throw new SyntaxErrorException
                      ("Unexpected Character Encountered: " + firstChar);
            }
        } // end loop.
        // Pop any remaining operators and
        // append them to postfix.
        while (!operatorStack.isEmpty()) {
            char op = operatorStack.pop();
            postfix.add(Character.toString(op));
        }
        // assert: Stack is empty, return.
    } catch (NoSuchElementException ex) {
        throw new SyntaxErrorException
            ("Syntax Error: The stack is empty");
    }
}

/** Method to process operators.
    @param op The operator
    @throws NoSuchElementException
 */
private void processOperator(char op) {
    if (operatorStack.isEmpty()) {
        operatorStack.push(op);
    } else {
         // Peek the operator stack and
         // let topOp be top operator.
         char topOp = operatorStack.peek();
         if (precedence(op) > precedence(topOp)) {
             operatorStack.push(op);
         }
```

```
                else {
                    // Pop all stacked operators with equal
                    // or higher precedence than op.
                    while (!operatorStack.isEmpty() && precedence(op) <=
                           precedence(topOp)) {
                        operatorStack.pop();
                        postfix.add(Character.toString(topOp));
                        if (!operatorStack.isEmpty()) {
                            // Reset topOp.
                            topOp = operatorStack.peek();
                        }
                    }
                    // assert: Operator stack is empty or
                    // current operator precedence >
                    // top of stack operator precedence.
                    operatorStack.push(op);
                }
            }
        }

        /** Determine whether a character is an operator.
            @param ch The character to be tested
            @return true if ch is an operator
         */
        private static boolean isOperator(char ch) {
            return OPERATORS.indexOf(ch) != -1;
        }

        /** Determine the precedence of an operator.
            @param op The operator
            @return the precedence
         */
        private static int precedence(char op) {
            return PRECEDENCE[OPERATORS.indexOf(op)] ;
        }
    }
```

PROGRAM STYLE

Updating a `StringJoiner` Is an Efficient Operation

We used a `StringJoiner` object for `postfix` because we knew that `postfix` was going to be continually updated and each token was separated by a space. Because `String` objects are immutable, it would have been less efficient to use a `String` object for `postfix`. A new `String` object would have to be allocated each time `postfix` changed.

Testing Listing 4.8 shows a `JUnit` test class for the `InfixToPostfix` class. Note that we are careful to type a space character between operands and operators.

We use enough test expressions to satisfy ourselves that the conversion is correct for properly formed input expressions. For example, try different orderings and multiple occurrences of the operators. You should also try infix expressions where all operators have the same precedence (e.g., all multiplicative).

If `convert` detects a syntax error, it will throw the exception `InfixToPostfix.SyntaxErrorException`. The driver will catch this exception and display an error message. If an exception is not thrown, the driver will display the result. Unfortunately, not all possible errors are detected. For example, an adjacent pair of operators or operands is not detected. To detect this error, we would need to add a `boolean` flag whose value indicates whether the last token was an operand. If the flag is `true`, the next token must be an operator; if the flag is `false`, the next token must be an operand. This modification is left as an exercise.

LISTING 4.8
`JUnit` test for `InfixToPostfix`

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

/**
 * Test for InfoxToPostfix
 * @author Koffman & Wolfgang
 */
public class InfixToPostfixTest {

    public InfixToPostfixTest() {
    }

    @Test
    public void simpleExpressionWithSamePrecedence() throws Exception {
        String infix = "a + b";
        String expResult = "a b +";
        String result = InfixToPostfix.convert(infix);
        assertEquals(expResult, result);
    }
    @Test
    public void simpleExpressionWithNumbersSamePrecedence() throws Exception {
         String infix = "2.5 * 6";
         String expResult = "2.5 6 *";
         String result = InfixToPostfix.convert(infix);
         assertEquals(expResult, result);
     }

    @Test
    public void expressionWithMixedPrecedence() throws Exception {
        String infix = "x1 - y / 2 + foo";
        String expResult = "x1 y 2 / - foo +";
        String result = InfixToPostfix.convert(infix);
        assertEquals(expResult, result);
    }

    @Test
    public void expressionWithInvalidOperator() throws Exception {
        String infix = "x1 & 2";
        Exception ex =
           assertThrows(InfixToPostfix.SyntaxErrorException.class,
             () -> InfixToPostfix.convert(infix));
        assertEquals("Unexpected Character Encountered: &", ex.getMessage());
    }

}
```

CASE STUDY Converting Expressions with Parentheses

Problem The ability to convert expressions with parentheses is an important (and necessary) addition. Parentheses are used to separate expressions into subexpressions.

Analysis We can think of an opening parenthesis on an operator stack as a boundary or fence between operators. Whenever we encounter an opening parenthesis, we want to push it onto the stack. A closing parenthesis is the terminator symbol for a subexpression. Whenever we encounter a closing parenthesis, we want to pop off all operators on the stack until we pop the patching opening parenthesis. Neither opening nor closing parentheses should appear in the postfix expression. Because operators scanned after the opening parenthesis should be evaluated before the opening parenthesis, the precedence of the opening parentheses must be smaller than any other operator. We also give a closing parenthesis the lowest precedence. This ensures that a "(" can only be popped by a ")".

Design We should modify method `processOperator` to `push` each opening parenthesis onto the stack as soon as it is scanned. Therefore, the method should begin as follows:

```
if (operatorStack.isEmpty() || op == '(') {
    operatorStack.push(op);
```

When a closing parenthesis is scanned, we want to pop all operators up to and including the matching opening parenthesis, inserting all operators popped (except for the opening parenthesis) in the postfix string. This will happen automatically in the `while` statement if the precedence of the closing parenthesis is smaller than that of any other operator except for the opening parenthesis:

```
while (!operatorStack.isEmpty() && precedence(op) <= precedence(topOp)) {
    operatorStack.pop();
    if (topOp == '(') {
        // Matching '(' popped - exit loop.
        break;
    }
    postfix.add(Character.toString(topOp));
```

A closing parenthesis is considered processed when an opening parenthesis is popped from the stack and the closing parenthesis is not placed on the stack. The following `if` statement executes after the `while` loop exits:

```
if (op != ')')
    operatorStack.push(op);
```

Implementation Listing 4.9 shows class `InfixToPostfixParens`, modified to handle parentheses. The additions are shown in bold. We have omitted parts that do not change.

Rather than impose the requirement of spaces between delimiters, we will use the `Scanner` method `findInLine` to extract tokens. The statements

```
var scan = new Scanner(infix);
while ((nextToken = scan.findInLine(PATTERN)) != null) {
```

create a `Scanner` object to scan the characters in `infix` (see Appendix A.10). The `while` loop repetition condition calls method `findInLine` to extract the next token from `infix`. The string constant `PATTERN` is a regular expression that describes the form of a token:

```
private static final String PATTERN =
  "\\d+\\.\\d*|\\d+|" + "\\p{L}[\\p{L}\\p{N}]*" + "|[" + OPERATORS + "]";
```

Each token can be an integer (a sequence of one or more digits), a double (a sequence of one or more digits followed by a decimal point followed by zero or more digits), an identifier (a letter followed by zero or more letters or digits), or an operator. The operators are processed in the same way as was done in Listing 4.7. Loop exit occurs after all tokens are extracted (`findInLine` returns null).

LISTING 4.9

InfixToPostfixParens.java

```
import java.util.*;

/** Translates an infix expression with parentheses
    to a postfix expression.
 */
public class InfixToPostfixParens {

    // Insert nested class SyntaxErrorException here. See Listing 4.6.

    // Data Fields
    /** The operators. */
    private static final String OPERATORS = "-+*/()";
    /** The precedence of the operators, matches order of OPERATORS. */
    private static final int[] PRECEDENCE = {1, 1, 2, 2, -1, -1};
    /** The Pattern to extract tokens.
        A token is either a number, an identifier, or an operator */
    private static final String PATTERN =
            "\\d+\\.\\d*|\\d+|" + "\\p{L}[\\p{L}\\p{N}]*" + "|[" + OPERATORS + "]";
    /** The stack of characters. */
    private final Deque<Character> operatorStack = new ArrayDeque<>();
    /** The postfix string. */
    private final StringJoiner postfix = new StringJoiner(" ");

    /** Convert a string from infix to postfix. Public convert is called
        by a client - Calls private method convertToPostfix to do the conversion.
        @param infix The infix expression
        @throws SyntaxErrorException
        @return the equivalent postfix expression.
     */
    public static String convert(String infix) throws SyntaxErrorException {
        infixToPostfixParens = new InfixToPostfixParens();
        infixToPostfixParens.convertToPostfix(infix);
        return infixToPostfixParens.getPostfix();
    }

    /** Returns the final postfix string.
     * @return the final postfix string
     */
    private String getPostfix() {
        return postfix.toString();
    }

    /** Convert a string from infix to postfix.
     *  Uses a stack to convert an infix expression to postfix
     *  pre: operator stack is empty
```

```
     *  post: postFix contains postfix expression and stack is empty
     *  @param infix the string to convert to postfix
     *  @throws SyntaxErrorException if argument is invalid
     */
    private void convertToPostfix(String infix) throws SyntaxErrorException {
        // Process each token in the infix string.
        try {
            String nextToken;
            var scan = new Scanner(infix);
            while ((nextToken = scan.findInLine(PATTERN)) != null) {
                char firstChar = nextToken.charAt(0);
                // Is it an operand?
                if (Character.isLetter(firstChar) || Character.isDigit(firstChar)) {
                    postfix.add(nextToken);
                } // Is it an operator?
                else if(isOperator(firstChar)) {
                    processOperator(firstChar);
                }
                else {
                    throw new SyntaxErrorException
                          ("Unexpected Character Encountered: " + firstChar);
                }
            } // end loop.
            // Pop any remaining operators
            // and append them to postfix.
            while (!operatorStack.isEmpty()) {
                char op = operatorStack.pop();
                // Any '(' on the stack is not matched.
                if (op == '(') throw new SyntaxErrorException
                                    ("Unmatched opening parenthesis");
                postfix.add(Character.toString(op));
            }
            // assert: Stack is empty, return result.
            return postfix.toString();
        } catch (NoSuchElementException ex) {
            throw new SyntaxErrorException("Syntax Error: The stack is empty");
        }
    }

    /** Method to process operators.
        @param op The operator
        @throws NoSuchElementException
     */
    private void processOperator(char op) {
        if (operatorStack.isEmpty() || op == '(') {
            operatorStack.push(op);
        }
        else {
            // Peek the operator stack and
            // let topOp be the top operator.
            char topOp = operatorStack.peek();
            if (precedence(op) > precedence(topOp)) {
                operatorStack.push(op);
            }
            else {
                 // Pop all stacked operators with equal
```

```
                    // or higher precedence than op.
                    while (!operatorStack.isEmpty() && precedence(op) <=
                           precedence(topOp)) {
                        operatorStack.pop();
                        if (topOp == '(') {
                            // Matching '(' popped - exit loop.
                            break;
                        }
                        postfix.add(new Character(topOp).toString());
                        if (!operatorStack.isEmpty()) {
                            // Reset topOp.
                            topOp = operatorStack.peek();
                        }
                    }

                    // assert: Operator stack is empty or
                    // current operator precedence >
                    // top of stack operator precedence.
                    if (op != ')')
                        operatorStack.push(op);
                }
            }
        }

        // Insert isOperator and precedence here. See Listing 4.7.
}
```

Tying the Case Studies Together

You can use the classes developed for the previous case studies to evaluate infix expressions with integer operands and nested parentheses. The argument for method `convert` will be the infix expression. The result will be its postfix form. Next, apply the method `PostfixEvaluator.eval`. The argument for `eval` will be the postfix expression returned by `convert`.

EXERCISES FOR SECTION 4.4

SELF-CHECK

1. Trace the evaluation of the following expressions using class `PostfixEvaluator`. Show the operand stack each time it is modified.

```
 4 7 + 5 * 3 - 6 /
13 2 * 5 / 6 2 5 * - +
 5 4. / 6 7 - 4 2 / - *
```

2. Trace the conversion of the following expressions to postfix using class `InfixToPostfix` or `InfixToPostfixParens`. Show the operator stack each time it is modified.

```
y - 7 / 35 + 4 * 6 - 10
( x + 15 ) * ( 3 * ( 4 - (5 + 7 / 2 ) )
```

PROGRAMMING

1. Modify class `InfixToPostfix` to handle the exponentiation operator, indicated by the symbol ^. The first operand is raised to the power indicated by the second operand. Assume that a sequence of ^ operators will not occur and that `precedence('^') > precedence('*')`.
2. Discuss how you would modify the infix-to-postfix `convert` method to detect a sequence of two operators or two operands.

4.5 Queue Abstract Data Type

You can think of a queue as a line of customers waiting for a scarce resource, such as a line waiting to buy tickets to an event. Figure 4.4 shows a line of "men" waiting to enter a restroom. The next one to enter the restroom is the one who has been waiting the longest, and latecomers are added to the end of the line. The `Queue` ADT gets its name from the fact that such a waiting line is called a "queue" in English-speaking countries other than the United States.

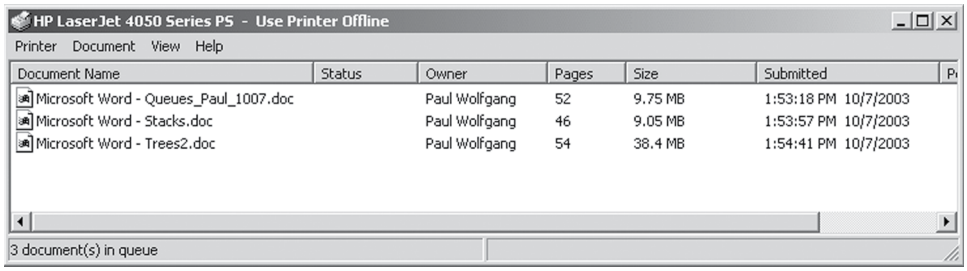
A Print Queue

In computer science, queues are used in operating systems to keep track of tasks waiting for a scarce resource and to ensure that the tasks are carried out in the order that they were generated. One example is a print queue. A Web surfer may select several pages to

FIGURE 4.4
Co-author Elliot Koffman is the next-to-last "person" in the queue.

Caryn J. Koffman

FIGURE 4.5
A Print Queue in the Windows Operating System

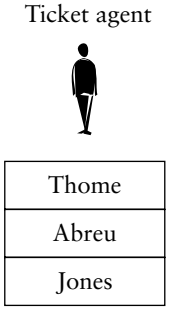


be printed in a few seconds. Because a printer is a relatively slow device (approximately 10 pages/minute), you will often select new pages to print faster than they can be printed. Rather than require you to wait until the current page is finished before you can select a new one, the operating system stores documents to be printed in a print queue (see Figure 4.5). Because they are stored in a queue, the pages will be printed in the same order as they were selected (first-in, first-out). The document first inserted in the queue will be the first one printed.

The Unsuitability of a "Print Stack"

Suppose your operating system used a stack (last-in, first-out) instead of a queue to store documents waiting to be printed. Then the most recently selected Web page would be the next page to be printed. This may not matter if only one person is using the printer. However, if the printer is connected to a computer network, this would be a big problem. Unless the print queue was empty when you selected a page to print (and the page printed immediately), that page would not print until all pages selected after it (by yourself or any other person on the network) were printed. If you were waiting by the printer for your page to print before going to your next class, you would have no way of knowing how long your wait might be. You would also be very unhappy if people who started after you had their documents printed before yours. So a print queue is a much more sensible alternative than a print stack.

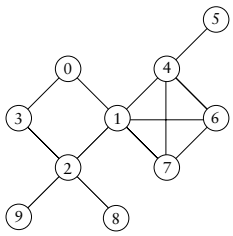
FIGURE 4.6
A Queue of Customers



A Queue of Customers

A queue of three customers waiting to buy concert tickets is shown in Figure 4.6. The name of the customer who has been waiting the longest is Thome; the name of the most recent arrival is Jones. Customer Thome will be the first customer removed from the queue (and able to buy tickets) when a ticket agent becomes available, and customer Abreu will then become the first one in the queue. Any new customers will be inserted in the queue after customer Jones.

FIGURE 4.7
A Network of Nodes



Using a Queue for Traversing a Multi-Branch Data Structure

In Chapter 10, you will see a data structure, called a graph, that models a *network* of *nodes*, with many links connecting each node to other nodes in the network (see Figure 4.7). Unlike a linked list, in which each node has only one successor, a node in a graph may have several successors. For example, node 0 in Figure 4.7 has nodes 1 and 3 as its successors. Consequently, it is not a simple matter to visit the nodes in a systematic way and to ensure that each node is visited only once. Programmers often use a queue to ensure that nodes closer to the starting point are visited before nodes that are farther away.

We will not go into the details here because we cover them later, but the idea is to put nodes that have not yet been visited into the queue when they are first encountered. After visiting the current node, the next node to visit is taken from the queue. This ensures that nodes are visited in the same order that they were encountered. Such a traversal is called a *breadth-first traversal* because the nodes visited spread out from the starting point. If we use a stack to hold the new nodes that are encountered and take the next node to visit from the stack, we will follow one path to the end before embarking on a new path. This kind of traversal is called a *depth-first traversal*.

Specification for a Queue Interface

The `java.util` API provides a `Queue` interface (Table 4.8) that extends the `Collection` interface and, therefore, the `Iterable` interface (see Table 2.34).

Method `offer` inserts an element at the rear of the queue and returns `true` if successful and `false` if unsuccessful. Methods `remove` and `poll` both remove and return the element at the front of the queue. The only difference in their behavior is when the queue happens to be empty: `remove` throws an exception and `poll` just returns `null`. Methods `peek` and `element` both return the element at the front of the queue without removing it. The difference is that `element` throws an exception when the queue is empty.

Because interface `Queue` extends interface `Collection`, a full implementation of the `Queue` interface must implement all required methods of the `Collection` interface. Classes that implement the `Queue` interface need to code the methods in Table 4.8 as well as methods `add`, `iterator`, `isEmpty`, and `size` declared in the `Collection` interface.

Class `LinkedList` Implements the Queue Interface

Because the `LinkedList` class provides methods for inserting and removing elements at either end of a double-linked list, all the `Queue` methods can be easily implemented in class `LinkedList`. For this reason, the `LinkedList` class implements the `Queue` interface. The statement

```
Queue<String> names = new LinkedList<>();
```

creates a new `Queue` reference, `names`, that stores references to `String` objects. The actual object referenced by `names` is type `LinkedList<String>`. However, because `names` is a type `Queue<String>` reference, you can apply only the `Queue` methods to it.

TABLE 4.8
Specification of Interface `Queue<E>`

Method	Behavior
`boolean offer(E item)`	Inserts `item` at the rear of the queue. Returns `true` if successful; returns `false` if the item could not be inserted
`E remove()`	Removes the entry at the front of the queue and returns it if the queue is not empty. If the queue is empty, throws a `NoSuchElementException`
`E poll()`	Removes the entry at the front of the queue and returns it; returns `null` if the queue is empty
`E peek()`	Returns the entry at the front of the queue without removing it; returns `null` if the queue is empty
`E element()`	Returns the entry at the front of the queue without removing it. If the queue is empty, throws a `NoSuchElementException`

EXAMPLE 4.2 We can use `jshell` which we introduced in Appendix A.1 to explore the queue operations. The shaded lines below show each `jshell` command; the result is below the command.

```
jshell> Queue<String> candidates = new LinkedList<>()
candidates ==> []
```

This command creates an empty queue `candidates`. The next commands execute the `offer` method four times.

```
jshell> candidates.offer("Pete")
$2 ==> true
jshell> candidates.offer("Amy")
$3 ==> true
jshell> candidates.offer("Bernie")
$4 ==> true
jshell> candidates.offer("Joe")
$5 ==> true
```

The `toString` method shows the queue contents where `Pete` is at the front and `Joe` is at the rear (Figure 4.8(a)).

```
jshell> candidates.toString()
$6 ==> "[Pete, Amy, Bernie, Joe]"
```

After two calls to `remove` shown below, the `peek` method shows that `Bernie` is now at the front of the queue and the `size` method shows that there are two elements left.

```
jshell> candidates.remove()
$7 ==> "Pete"
jshell> candidates.remove()
$8 ==> "Amy"
jshell> candidates.peek()
$9 ==> "Bernie"
jshell> candidates.size()
$10 ==> 2
jshell> candidates.toString()
$11 ==> "[Bernie, Joe]"
```

The last statement shows the queue contents (Figure 4.8(b)).

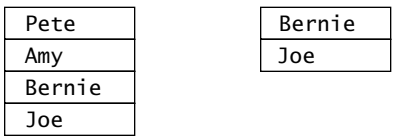


FIGURE 4.8 Queue Candidates

EXERCISES FOR SECTION 4.5

SELF-CHECK

1. Draw the queue in Figure 4.8(a) as it will appear after the insertion of customer Harris and Jones and the removal of one candidate from the queue. Which candidate is removed? How many candidates are left?
2. What would be shown after each of the commands below executes:

```
jshell> candidates.poll();
jshell> candidates.size();
jshell> candidates.toString();
```

3. Assume that myQueue is an instance of a class that implements Queue<String> and myQueue is an empty queue. Explain the effect of each of the following operations.

```
myQueue.offer("Hello");
myQueue.offer("Bye");
myQueue.offer("Ciao");
System.out.println(myQueue.peek());
myQueue.remove();
myQueue.offer("Welcome");
myQueue.remove();
if (!myQueue.isEmpty()) {
    System.out.println(myQueue.remove() + ", new size is " + myQueue.size());
    System.out.println("Item in front is " + myQueue.peek());
}
```

4. For the queue names in Figure 4.6, what is the effect of the following?

```
while (!names.isEmpty()) {
    System.out.println(names.remove());
}
```

5. What would be the effect of using peek instead of remove in Question 4?

PROGRAMMING

1. Write a main method that creates two stacks of Integer objects and a queue of Integer objects. Store the numbers –1, 15, 23, 44, 4, 99 in the first stack. The top of the stack should store 99.
2. Write a loop to get each number from the first stack and store it in the second stack and in the queue.
3. Write a second loop to remove a value from the second stack and from the queue and display each pair of values on a separate output line. Continue until the data structures are empty. Show the output.

4.6 Queue Applications

In this section we present an application that maintains a queue of Strings representing the names of customers waiting for service. Our goal is just to maintain the list and ensure that customers are inserted and removed properly. We will allow a user to determine the queue size, the person at the front of the queue, and how many people are ahead of a particular person in the queue.

CASE STUDY Maintaining a Queue

Problem Write a menu-driven program that maintains a list of customers waiting for service. The program user should be able to insert a new customer in the line, display the customer who is next in line, remove the customer who is next in line, display the length of the line, or determine how many people are ahead of a specified customer.

Analysis As discussed earlier, a queue is a good data structure for storing a list of customers waiting for service because they would expect to be served in the order in which they arrived. We can display the menu and then perform the requested operation by calling the appropriate Queue method to update the customer list.

TABLE 4.9
Class `MaintainQueue`

Method	Behavior
`public static void processCustomers()`	Accepts and processes each user's selection

Problem Inputs

The operation to be performed
The name of a customer

Problem Outputs

The effect of each operation

Design

We will write a class `MaintainQueue` to store the queue and control its processing. Class `MaintainQueue` has a `Queue<String>` component `customers`.

Method `processCustomers` displays a menu of choices and processes the user selection by calling the appropriate `Queue` method. Table 4.9 shows class `MaintainQueue`.

The algorithm for method `processCustomers` follows.

Algorithm for `processCustomers`

1. `while` the user is not finished
2. Display the menu and get the operation selected.
3. Perform the operation selected.

Each operation is performed by a call to one of the `Queue` methods, except for determining the position of a particular customer in the queue. The algorithm for this operation follows.

Algorithm for Determining the Position of a Customer

1. Get the customer name.
2. Set the count of customers ahead of this one to 0.
3. for each customer in the queue
4. `if` this customer is not the one sought
5. Increment the count.
6. else
7. Display the count of customers and exit the loop.
8. `if` all customers were examined without success
9. Display a message that the customer is not in the queue.

The loop that begins at step 3 requires us to access each element in the queue. However, only the element at the front of the queue is directly accessible using method `peek` or `element`. We will show how to get around this limitation by using an `Iterator` to access each element of the queue.

Implementation

Listing 4.10 shows the data field declarations and the constructor for class `MaintainQueue`. The constructor sets `customers` to reference an instance of class `LinkedList<String>`.

LISTING 4.10

Constructor for Class `MaintainQueue`

```
import java.util.Queue;
import java.util.LinkedList;
import java.util.NoSuchElementException;
import java.util.Scanner;
import java.util.Arrays;
/**
 *  Class to maintain a queue of customers.
 *  @author Koffman & Wolfgang
 */
public class MaintainQueue {

    // Data Field
    private final Queue<String> customers;
    private final Scanner in;

  // Constructor
  /** Create an empty queue. */
  public MaintainQueue() {
      customers = new LinkedList<>();
      in = new Scanner(System.in);
    }
```

In method `processCustomers` (Listing 4.11), the `while` loop executes until the user enters `"quit"`. The user enters each desired operation into `choice`. The switch statement calls a `Queue` method to perform the selected operation. For example, if the user enters `"add"`, the following statements read the customer name and insert it into the queue.

```
System.out.println("Enter new customer name");
name = in.nextLine();
customers.offer(name);
```

Case `"position"` finds the position of a customer in the queue. The enhanced `for` statement uses an `Iterator` to access each element of the queue and store it in `nextName`. The `if` condition compares `nextName` to the name of the customer being sought. The variable `countAhead` is incremented each time this comparison is unsuccessful.

```
int countAhead = 0;
for (String nextName : customers) {
    if (!nextName.equals(name)) {
        countAhead ++;
    } else {
        System.out.println("The number of customers ahead of " + name
                           + " is " + countAhead);
        break; // Customer found, exit loop.
    }
}
```

If the desired name is accessed, its position is displayed and the loop is exited. If the name is not found, the loop is exited after the last name is processed. The `if` statement following the loop displays a message if the name was not found. This will be the case when `countAhead` is equal to the queue size.

The `switch` statement is inside a `try-catch` sequence that handles a `NoSuchElementException` (caused by an attempt to remove or retrieve an element from an empty queue) by displaying an error message.

LISTING 4.11

Method `processCustomers` in Class `MaintainQueue`

```
/*
 *  Performs the operations selected on queue customers.
 *  pre: customers has been created.
 *  post: customers is modified based on user selections.
 */
public void processCustomers() {
    String choice =
    String[] choices =
        {"add", "peek", "remove", "size", "position", "quit"};
    // Perform all operations selected by user.
    while (!choice.equals("quit")) {
        // Process the current choice.
        try {
            String name;
            System.out.println("Choose from the list: "
                               + Arrays.toString(choices));
            choice = in.nextLine();
            switch (choice) {
                case "add":
                    System.out.println("Enter new customer name");
                    name = in.nextLine();
                    customers.offer(name);
                    System.out.println("Customer " + name +
                                       " added to the queue");
                    break;
                case "peek":
                    System.out.println("Customer " + customers.element() +
                                       " is next in the queue");
                    break;
                case "remove":
                    System.out.println("Customer " + customers.remove() +
                                       " removed from the queue");
                    break;
                case "size":
                    System.out.println("Size of queue is " + customers.size());
                    break;
                case "position":
                    System.out.println("Enter customer name");
                    name = in.nextLine();
                    int countAhead = 0;
                    for (String nextName : customers) {
                        if (!nextName.equals(name)) {
                            countAhead++;
                        } else {
                            System.out.println("The number of customers ahead of "
                                               + name + " is " + countAhead);
                            break;
                            // Customer found, exit loop.
                        }
                    }

                    // Check whether customer was found.
                    if (countAhead == customers.size()) {
                        System.out.println(name + " is not in queue");
                    }
```

```
                    break;
                case "quit":
                    System.out.println("Leaving customer queue. "
                                       + "\nNumber of customers in queue is "
                                       + customers.size());
                    break;
                default:
                    System.out.println("Invalid choice - try again");
                } // end switch
            } catch (NoSuchElementException e) {
                System.out.println("The Queue is empty");
            } // end try-catch
        } // end while
    }
```

Testing

You can use class `MaintainQueue` to test each of the different `Queue` implementations discussed in the next section. You should verify that all customers are stored and retrieved in FIFO order. You should also verify that a `NoSuchElementException` is thrown if you attempt to remove or retrieve a customer from an empty queue. Thoroughly test the queue by selecting different sequences of queue operations. Figure 4.9 shows a sample run.

FIGURE 4.9 Sample Run of Client of `MaintainQueue`

```
Choose from the list: [add, peek, remove, size, position, quit]
add
Enter new customer name
koffman
Customer koffman added to the queue
Choose from the list: [add, peek, remove, size, position, quit]
add
Enter new customer name
wolfgang
Customer wolfgang added to the queue
Choose from the list: [add, peek, remove, size, position, quit]
remove
Customer koffman removed from the queue
Choose from the list: [add, peek, remove, size, position, quit]
peel
Invalid choice - try again
Choose from the list: [add, peek, remove, size, position, quit]
peek
Customer wolfgang is next in the queue
Choose from the list: [add, peek, remove, size, position, quit]
quit
Leaving customer queue.
Number of customers in queue is 1
```

PROGRAM STYLE

When to Use the Different Queue Methods

For a queue of unlimited size, `add` and `offer` are logically equivalent. Both will return `true` and never throw an exception. For a bounded queue, `add` will throw an exception if the `queue` is full, but `offer` will return `false`.

For `peek` versus `element` and `poll` versus `remove`, `peek` and `poll` don't throw exceptions, but the user should either check for a return value of `null`, or be sure that the calls are within an `if` or `while` block that tests for a nonempty queue before they are called.

Using Queues for Simulation

Simulation is a technique used to study the performance of a physical system by using a physical, mathematical, or computer model of the system. Through simulation, the designers of a new system can estimate the expected performance of the system before they actually build it. The use of simulation can lead to changes in the design that will improve the expected performance of the new system. Simulation is especially useful when the actual system would be too expensive to build or too dangerous to experiment with after its construction.

The textbook website provides a case study of a computer simulation of an airline check-in counter in order to compare various strategies for improving service and reducing the waiting time for each passenger. It uses a queue to simulate the passenger waiting line. A special branch of mathematics called *queuing theory* has been developed to study these kinds of problems using mathematical models (systems of equations) instead of computer models.

EXERCISES FOR SECTION 4.6

SELF-CHECK

1. Write an algorithm to display all the elements in a queue using just the queue operations. How would your algorithm change the queue?

2. Trace the following fragment for a `Stack<String> s` and an empty queue `q` (type `Queue<String>`).

```
String item;
while (!s.empty()) {
    item = s.pop();
    q.offer(item);
}
while (!q.isEmpty()) {
    item = q.remove();
    s.push(item);
}
```

   a. What is stored in stack `s` after the first loop executes? What is stored in queue `q` after the first loop executes?

   b. What is stored in stack `s` after the second loop executes? What is stored in queue `q` after the second loop executes?

PROGRAMMING

1. Write a `toString` method for class `MaintainQueue`.

4.7 Implementing the Queue Interface

In this section we discuss three approaches to implementing a queue: using a double-linked list, a single-linked list, and an array. We begin with a double-linked list.

Using a Double-Linked List to Implement the Queue Interface

Insertion and removal from either end of a double-linked list is O(1), so either end can be the front (or rear) of the queue. The Java designers decided to make the head of the linked list the front of the queue and the tail the rear of the queue. If you declare your queue using the statement:

```
Queue<String> myQueue = new LinkedList<>();
```

the fact that the actual class for `myQueue` is a `LinkedList` is not visible. The only methods available for `myQueue` are those declared in the `Queue` interface.

Using a Single-Linked List to Implement the Queue Interface

We can implement a queue using a single-linked list like the one shown in Figure 4.10. Class `ListQueue` contains a collection of `Node<E>` objects (see Section 2.5). Recall that class `Node<E>` has attributes `data` (type `E`) and `next` (type `Node<E>`).

Insertions are at the rear of a queue, and removals are from the front. We need a reference to the last list node so that insertions can be performed in O(1) time; otherwise, we would have to start at the list head and traverse all the way down the list to do an insertion. There is a reference variable `front` to the first list node (the list head) and a reference variable `rear` to the last list node. There is also a data field `size`.

The number of elements in the queue is changed by methods `insert` and `remove`, so `size` must be incremented by one in `insert` and decremented by one in `remove`. The value of `size` is tested in `isEmpty` to determine the status of the queue. The method `size` simply returns the value of data field `size`.

Listing 4.12 shows class `ListQueue<E>`. Method `offer` treats insertion into an empty queue as a special case because both `front` and `rear` should reference the new node after the insertion.

```
rear = new Node<>(item); front = rear;
```

If we insert into a queue that is not empty, the new node must be linked to the old rear of the queue, but `front` is unchanged.

```
rear.next = new Node<>(item);
rear = rear.next;
```

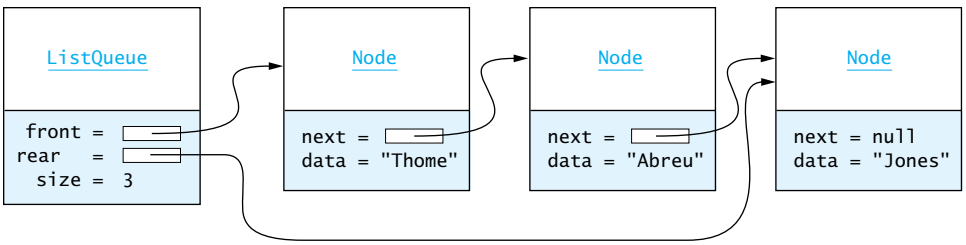


FIGURE 4.10
A Queue as a Single-Linked List

If the queue is empty, method peek returns null. Otherwise, it returns the element at the front of the queue:

```
return front.data;
```

Method poll calls method peek and returns its result. However, before returning, it disconnects the node at the front of a nonempty queue and decrements size.

```
front = front.next;
size--;
```

Listing 4.12 is incomplete. To finish it, you need to write methods remove, element, size, and isEmpty. You also need to code an iterator method and a class Iter with methods next, hasNext, and remove. This class will be similar to class KWListIter (see Section 2.6).

You can simplify this task by having ListQueue<E> extend class java.util.AbstractQueue<E>. This class implements add, remove, and element using offer, poll, and peek, and inherits from its superclass, AbstractCollection<E>, all methods needed to implement the Collection<E> interface (the superinterface of Queue<E>).

LISTING 4.12
Class ListQueue

```
import java.util.*;
/** Implements the Queue interface using a single-linked list. */
public class ListQueue<E> extends AbstractQueue<E>
             implements Queue<E> {

    // Data Fields
    /** Reference to front of queue. */
    private Node<E> front;
    /** Reference to rear of queue. */
    private Node<E> rear;
    /** Size of queue. */
    private int size;

    // Insert inner class Node<E> for single-linked list here.
    // (See Listing 2.1.)
    // Methods
    /** Insert an item at the rear of the queue.
        post: item is added to the rear of the queue.
        @param item The element to add
        @return true (always successful)
     */
    @Override
    public boolean offer(E item) {
        // Check for empty queue.
        if (front == null) {
            rear = new Node<>(item);
            front = rear;
        } else {
            // Allocate a new node at end, store item in it, and
            // link it to old end of queue.
            rear.next = new Node<>(item);
            rear = rear.next;
        }
        size++;
        return true;
    }

    /** Remove the entry at the front of the queue and return it
        if the queue is not empty.
```

```
        post: front references item that was second in the queue.
        @return The item removed if successful, or null if not
     */
    @Override
    public E poll() {
        // Retrieve item at front.
        E item = peek();
        if (item == null)
            return null;
        // Remove item at front.
        front = front.next; size--;
        // Return data at front of queue.
        return item;
    }

    /** Return the item at the front of the queue without removing it.
        @return The item at the front of the queue if successful;
                return null if the queue is empty
    */
    @Override
    public E peek() {
        if (size == 0)
            return null;
        else
            return front.data;
    }
    // Insert class Iter and other required methods.
    . . .
}
```

Using a Circular Array to Implement the Queue Interface

While the time efficiency of using a single- or double-linked list to implement the `Queue` interface is acceptable, there is some space inefficiency. Each node of a single-linked list contains a reference to its successor, and each node of a double-linked list contains references to its predecessor and successor. These additional references will increase the storage space required.

An alternative is to use an array. If we use an array, we can do an insertion at the rear of the array in O(1) time. However, a removal from the front will be an O(n) process if we shift all the elements that follow the first one over to fill the space vacated. Similarly, removal from the rear is O(1), but insertion at the front is O(n). We next discuss how to avoid this inefficiency.

Overview of the Design

To represent a queue, we will use an object with four `int` data members (`front`, `rear`, `size`, `capacity`), a constant DEFAULT_CAPACITY, and an array data member, `theData`, which provides storage for the queue elements.

```
    /** Index of the front of the queue. */
    private int front;
    /** Index of the rear of the queue. */
    private int rear;
    /** Current number of elements in the queue. */
    private int size;
    /** Current capacity of the queue. */
    private int capacity;
    /** Default capacity of the queue. */
    private static final int DEFAULT_CAPACITY = 10;

    /** Array to hold the data. */
    private E[] theData;
```

The `int` fields `front` and `rear` are indexes to the queue elements at the front and rear of the queue, respectively. The `int` field `size` keeps track of the actual number of items in the queue and allows us to determine easily whether the queue is empty (`size` is 0) or full (`size` is `capacity`).

It makes sense to store the first queue item in element 0, the second queue item in element 1, and so on. So we should set `front` to 0 and `rear` to –1 when we create an initially empty queue. Each time we do an insertion, we should increment `size` and `rear` by 1 so that `front` and `rear` will both be 0 if a queue has one element. Figure 4.11 shows an instance of a queue that is filled to its capacity (`size` is `capacity`). The queue contains the symbols `&`, `*`, `+`, `/`, `-`, inserted in that order.

Because the queue in Figure 4.11 is filled to capacity, we cannot insert a new character without allocating more storage. However, we can remove a queue element by decrementing `size` and incrementing `front` to 1, thereby removing `theData[0]` (the symbol `&`) from the queue. Figure 4.12 shows the queue after removing the first element (it is still in the array, but not part of the queue). The queue contains the symbols `*`, `+`, `/`, `-` in that order.

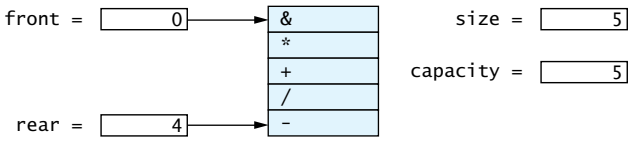


FIGURE 4.11
A Queue Filled with Characters

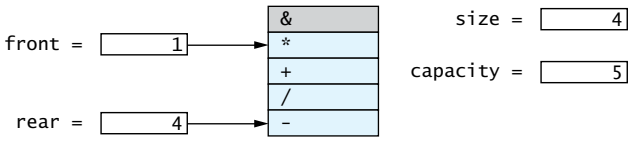


FIGURE 4.12
The Queue after Deletion of the First Element

Although the queue in Figure 4.12 is no longer filled, we cannot insert a new character because `rear` is at its maximum value. One way to solve the problem is to shift the elements in array `theData` so that the empty cells come after `rear` and then adjust `front` and `rear` accordingly. This array shifting must be done very carefully to avoid losing track of active array elements. It is also an O(n) operation.

A better way to solve this problem is to represent the array field `theData` as a *circular array*. In a circular array, the elements wrap around so that the first element actually follows the last. This is like counting modulo `size`; the array subscripts take on the values `0`, `1`, `...`, `size - 1`, `0`, `1`, and so on. This allows us to "increment" `rear` to 0 and store a new character in `theData[0]`. Figure 4.13 shows the queue after inserting a new element (the character `A`). After the insertion, `front` is still 1 but `rear` becomes 0. The contents of `theData[0]` change from `&` to `A`. The queue now contains the symbols `*`, `+`, `/`, `-`, `A` in that order.

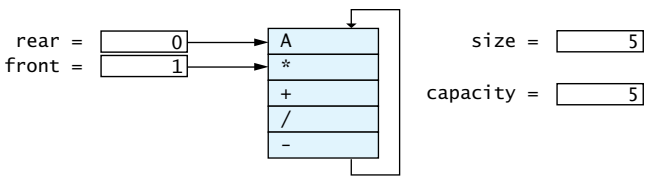


FIGURE 4.13
A Queue as a Circular Array

EXAMPLE 4.3 The upper half of Figure 4.14 shows the effect of removing two elements from the queue just described. There are currently three characters in this queue (stored in `theData[3]`, `theData[4]`, and `theData[0]`). The queue now contains the symbols `/`, `-`, `A` in that order.

The lower half of Figure 4.14 shows the queue after insertion of a new character (`B`). The value of `rear` is incremented to 1, and the next element is inserted in `theData[1]`. This queue element follows the character A in `theData[0]`. The value of `front` is still 3 because the character / at `theData[3]` has been in the queue the longest. `theData[2]` is now the only queue element that is unused. The queue now contains the symbols `/`, `-`, `A`, `B` in that order.

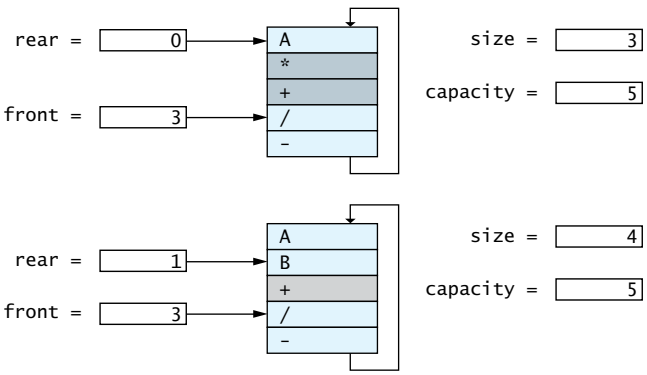


FIGURE 4.14 The Effect of Two Deletions . . . and One Insertion

Implementing ArrayQueue<E>

Listing 4.13 shows the implementation of the class `ArrayQueue<E>`.

The constructors set `size` to 0 and `front` to 0 because array element `theData[0]` is considered the front of the empty queue, and `rear` is initialized to `capacity - 1` (instead of –1) because the queue is circular.

In method `offer`, the statement

```
rear = (rear + 1) % capacity;
```

is used to increment the value of `rear` modulo `capacity`. When `rear` is less than `capacity`, this statement simply increments its value by one. But when `rear` becomes equal to `capacity - 1`, the next value of `rear` will be 0 (`capacity` mod `capacity` is 0), thereby wrapping the last element of the queue around to the first element. Because the constructor initializes `rear` to `capacity - 1`, the first queue element will be placed in `theData[0]` as desired.

In method `poll`, the element currently stored in `theData[front]` is copied into `result` before `front` is incremented modulo `capacity`; `result` is then returned. In method `peek`, the element at `theData[front]` is returned, but `front` is not changed.

LISTING 4.13
ArrayQueue.java

```
/** Implements the Queue interface using a circular array. */
public class ArrayQueue<E> extends AbstractQueue<E>
             implements Queue<E> {

    // Data Fields
    /** Index of the front of the queue. */
```

```
private int front;
/** Index of the rear of the queue. */
private int rear;
/** Current size of the queue. */
private int size;
/** Current capacity of the queue. */
private int capacity;
/** Default capacity of the queue. */
private static final int DEFAULT_CAPACITY = 10;
/** Array to hold the data. */
private E[] theData;

// Constructors
/** Construct a queue with the default initial capacity. */
public ArrayQueue() {
    this(DEFAULT_CAPACITY);
}

@SuppressWarnings("unchecked")
/** Construct a queue with the specified initial capacity.
    @param initCapacity The initial capacity
 */
public ArrayQueue(int initCapacity) {
    capacity = initCapacity;
    theData = (E[]) new Object[capacity];
    front = 0;
    rear = capacity - 1;
    size = 0;
}

// Public Methods
/** Inserts an item at the rear of the queue.
    post: item is added to the rear of the queue.
    @param item The element to add
    @return true (always successful)
 */
@Override
public boolean offer(E item) {
    if (size == capacity) {
        reallocate();
    }
    size++;
    rear = (rear + 1) % capacity; theData[rear] = item;
    return true;
}

/** Returns the item at the front of the queue without removing it.
    @return The item at the front of the queue if successful; return null if
            the queue is empty

 */
@Override
public E peek() {
    if (size == 0)
        return null;
    else
        return theData[front];
}

/** Removes the entry at the front of the queue and returns it if the queue is
    not empty.
```

```
        post: front references item that was second in the queue.
        @return The item removed if successful or null if not
     */
    @Override
    public E poll() {
        if (size == 0) {
            return null;
        }
        E result = theData[front];
        front = (front + 1) % capacity;
        size--;
        return result;
    }

    // Private Methods
    /** Double the capacity and reallocate the data.
        pre: The array is filled to capacity.
        post: The capacity is doubled and the first half of the expanded array is
              filled with data.
     */
    @SuppressWarnings("unchecked")
    private void reallocate() {
        int newCapacity = 2 * capacity;
        E[] newData = (E[]) new Object[newCapacity];
        int j = front;
        for (int i = 0; i < size; i++) {
            newData[i] = theData[j];
            j = (j + 1) % capacity;
        }
        front = 0;
        rear = size - 1;
        capacity = newCapacity;
        theData = newData;
    }
}
```

Increasing Queue Capacity

When the capacity is reached, we double the capacity and copy the array into the new one, as was done for the `ArrayList`. However, we can't simply use the `reallocate` method we developed for the `ArrayList` because of the circular nature of the array. We can't copy over elements from the original array to the first half of the expanded array, maintaining their position. We must first copy the elements from position `front` through the end of the original array to the beginning of the expanded array; then copy the elements from the beginning of the original array through `rear` to follow those in the expanded array (see Figure 4.15).

We begin by creating an array `newData`, whose capacity is double that of `theData`. The loop

```
int j = front;
for (int i = 0; i < size; i++) {
     newData[i] = theData[j];
     j = (j + 1) % capacity;
}
```

copies `size` elements over from `theData` to the first half of `newData`. In the copy operation

```
newData[i] = theData[j]
```

subscript `i` for `newData` goes from `0` to `size - 1` (the first half of `newData`). Subscript `j` for `theData` starts at `front`. The statement

```
j = (j + 1) % capacity;
```

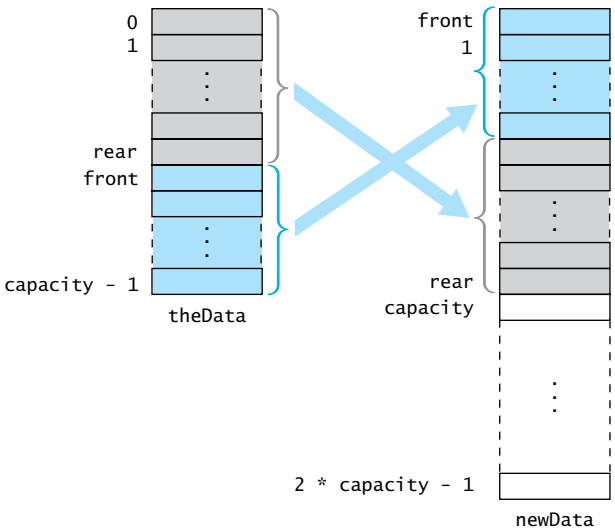


FIGURE 4.15
Reallocating a Circular Array

increments the subscript for array `theData`. Therefore, subscript `j` goes from `front` to `capacity - 1` (in increments of 1) and then back to 0. So the elements are copied from `theData` in the sequence `theData[front], ..., theData[capacity - 1], theData[0], ..., theData[rear]`, where `theData[front]` is stored in `newData[0]` and `theData[rear]` is stored in `newData[size - 1]`. After the copy loop, front is reset to 0 and rear is reset to size - 1 (see Figure 4.15).

By choosing a new capacity that is twice the current capacity, the cost of the reallocation is amortized across each insert, just as for an `ArrayList`. Thus, insertion is still considered an O(1) operation.

PITFALL

Incorrect Use of `Arrays.copyOf` to Expand a Circular Array

You might consider using the following method to copy all of the elements over from the original array `theData` to the first half of the expanded array `theData`.

```
private void reallocate() {
    capacity = 2 * capacity;
    theData = Arrays.copyOf(theData, capacity);
}
```

The problem is that in the circular array before expansion, element `theData[0]` followed the last array element. However, after expansion, the element that was formerly in the last position would now be in the middle of the array, so `theData[0]` would not follow it.

Implementing Class `ArrayQueue<E>.Iter`

Just as for class `ListQueue<E>`, we must implement the missing `Queue` methods and an inner class `Iter` to fully implement the `Queue` interface. Listing 4.14 shows inner class `Iter`.

Data field `index` stores the subscript of the next element to access. The constructor initializes `index` to `front` when a new `Iter` object is created. Data field `count` keeps track of the number of items accessed so far. Method `hasNext` returns `true` if `count` is less than the queue size.

LISTING 4.14

Class ArrayQueue<E>.Iter

```
/** Inner class to implement the Iterator<E> interface. */
private class Iter implements Iteratorxf<E> {
    // Data Fields
    // Index of next element
    private int index;
    // Count of elements accessed so far
    private int count = 0;

    // Methods
    // Constructor

    /** Initializes the Iter object to reference the first queue element. */
     public Iter() {
        index = front;
    }

    /** Returns true if there are more elements in the queue to access. */
    @Override
    public boolean hasNext() {
        return count < size;
    }

    /** Returns the next element in the queue.
        pre: index references the next element to access.
        post: index and count are incremented.
        @return The element with subscript index
     */
    @Override
    public E next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        E returnValue = theData[index];
        index = (index + 1) % capacity;
        count++;
        return returnValue;
    }

    /** Remove the item accessed by the Iter object - not implemented. */
    @Override
    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Method `next` returns the element at position `index` and increments `index` and `count`. Method `Iter.remove` throws an `UnsupportedOperationException` because it would violate the contract for a queue to remove an item other than the first one.

Comparing the Three Implementations

As mentioned earlier, all three implementations of the `Queue` interface are comparable in terms of computation time. All operations are O(1) regardless of the implementation. Although reallocating an array is an O(n) operation, it is amortized over n items, so the cost per item is O(1).

In terms of storage requirements, both linked-list implementations require more storage because of the extra space required for links. To perform an analysis of the storage requirements, you need to know that Java stores a reference to the data for a queue element in each node in addition to the links. Therefore, each node for a single-linked list would store a total of two references (one for the data and one for the link), a node for a double-linked list would store a total of three references, and a node for a circular array would store just one reference. Therefore, a double-linked list would require 1.5 times the storage required for a single-linked list with the same number of elements. A circular array that is filled to capacity would require half the storage of a single-linked list to store the same number of elements. However, if the array were just reallocated, half the array would be empty, so it would require the same storage as a single-linked list.

EXERCISES FOR SECTION 4.7

SELF-CHECK

1. Show the new array for the queue in Figure 4.13 after the array size is doubled.
2. Provide the algorithm for the methods in Programming Exercise 1 below.
3. Redraw the queue in Figure 4.10 so that `rear` references the list head and `front` references the list tail. Show the queue after an element is inserted and an element is removed. Explain why the approach used in the book is better.

PROGRAMMING

1. Write the missing methods required by the `Queue` interface and inner class `Iter` for class `ListQueue<E>`. Class `Iter` should have a data field `current` of type `Node<E>`. Data field `current` should be initialized to `first` when a new `Iter` object is created. Method `next` should return the value of `current` and advance `current`. Method `remove` should throw an `UnsupportedOperationException`.
2. Write the missing methods for class `ArrayQueue<E>` required by the `Queue` interface.
3. Replace the loop in method `reallocate` with two calls to `System.arraycopy`.

4.8 The Deque Interface

As we mentioned in Section 4.2, Java provides the `Deque` interface. The name deque (pronounced `"deck"`) is short for double-ended queue, which means that it is a data structure that allows insertions and removals from both ends (front and rear). Methods are provided to insert, remove, and examine elements at both ends of the deque. Method names that end in *first* access the front of the deque, and method names that end in *last* access the rear of the deque. Table 4.10 shows some of the `Deque` methods.

As you can see from the table, there are two pairs of methods that perform each of the insert, remove, and examine operations. One pair returns a `boolean` value indicating the method result, and the other pair throws an exception if the operation is unsuccessful. For example, `offerFirst` and `offerLast` return a value indicating the insertion result, whereas `addFirst`

TABLE 4.10
The `Deque<E>` Interface

Method	Behavior
`boolean offerFirst(E item)`	Inserts `item` at the front of the deque. Returns `true` if successful; returns `false` if the item could not be inserted
`boolean offerLast(E item)`	Inserts `item` at the rear of the deque. Returns `true` if successful; returns `false` if the item could not be inserted
`void addFirst(E item)`	Inserts `item` at the front of the deque. Throws an exception if the item could not be inserted
`void addLast(E item)`	Inserts `item` at the rear of the deque. Throws an exception if the item could not be inserted
`E pollFirst( )`	Removes the entry at the front of the deque and returns it; returns `null` if the deque is empty
`E pollLast( )`	Removes the entry at the rear of the deque and returns it; returns `null` if the deque is empty
`E removeFirst( )`	Removes the entry at the front of the deque and returns it if the deque is not empty. If the deque is empty, throws a `NoSuchElementException`
`E removeLast( )`	Removes the item at the rear of the deque and returns it. If the deque is empty, throws a `NoSuchElementException`
`E peekFirst( )`	Returns the entry at the front of the deque without removing it; returns `null` if the deque is empty
`E peekLast( )`	Returns the item at the rear of the deque without removing it; returns `null` if the deque is empty
`E getFirst( )`	Returns the entry at the front of the deque without removing it. If the deque is empty, throws a `NoSuchElementException`
`E getLast( )`	Returns the item at the rear of the deque without removing it. If the deque is empty, throws a `NoSuchElementException`
`boolean removeFirstOccurrence (Object item)`	Removes the first occurrence of item in the deque. Returns `true` if the item was removed
`boolean removeLastOccurrence(Object item)`	Removes the last occurrence of item in the deque. Returns `true` if the item was removed
`Iterator<E> iterator( )`	Returns an `iterator` to the elements of this deque in the proper sequence
`Iterator<E> descendingIterator( )`	Returns an `iterator` to the elements of this deque in reverse sequential order

and `addLast` throw an exception if the insertion is not successful. Normally, you should use a method that returns a value. Table 4.11 shows the use of these methods.

Classes that Implement Deque

The Java Collections Framework provides four implementations of the `Deque` interface, including `ArrayDeque` and `LinkedList`. `ArrayDeque` utilizes a resizable circular array like our

TABLE 4.11
Effect of Using `Deque` Methods on an Initially Empty `Deque<Character> d`.

Deque Method	Deque d	Effect
`d.offerFirst('b')`	`b`	`'b'` inserted at `front`
`d.offerLast('y')`	`by`	`'y'` inserted at `rear`
`d.addLast('z')`	`byz`	`'z'` inserted at `rear`
`d.addFirst('a')`	`abyz`	`'a'` inserted at `front`
`d.peekFirst()`	`abyz`	Returns `'a'`
`d.peekLast( )`	`abyz`	Returns `'z'`
`d.pollLast()`	`aby`	Removes `'z'`
`d.poUFirst()`	`by`	Removes `'a'`

class `ArrayQueue` and is the recommended implementation because, unlike `LinkedList`, it does not support indexed operations.

Using a Deque as a Queue

The `Deque` interface extends the `Queue` interface, which means that a class that implements `Deque` also implements `Queue`. The `Queue` methods are equivalent to `Deque` methods, as shown in Table 4.12. If elements are always inserted at the front of a deque and removed from the rear (FIFO), then the deque functions as a queue. In this case, you could use either method `add` or `addLast` to insert a new item.

TABLE 4.12
Equivalent `Queue` and `Deque` Methods

Queue Method	Equivalent Deque Method
`add(e)`	`addLast(e)`
`offer(e)`	`offerLast(e)`
`remove()`	`removeFirst()`
`poll()`	`pollFirst()`
`element()`	`getFirst()`
`peek()`	`peekFirst()`

Using a Deque as a Stack

Earlier in this chapter, we used deques as stacks (LIFO). When a deque is used as a stack, elements are always pushed and popped from the front of the deque. Using the `Deque` interface is preferable to using the legacy `Stack` class (based on the `Vector` class). Stack methods are equivalent to `Deque` methods, as shown in Table 4.13.

TABLE 4.13
Equivalent `Stack` and `Deque` Methods

Stack Method	Equivalent Deque Method
`push(e)`	`addFirst(e)`
`pop()`	`removeFirst()`
`peek()`	`peekFirst()`
`isEmpty()`	`isEmpty()`

The statement

```
Deque<String> stackOfStrings = new ArrayDeque<>();
```

creates a new `Deque` object called `stackOfStrings`. You can use methods `push`, `pop`, `peek`, and `isEmpty` in the normal way to manipulate `stackOfStrings`.

EXERCISES FOR SECTION 4.8

SELF-CHECK

1. For object `stackOfStrings` declared above, replace each stack operation with the appropriate `Deque` method and explain the effect of each statement in the following fragment.

```
stackOfStrings.push("Hello");
stackOfStrings.push("Welcome");
String one = stackOfStrings.pop();
if (!stackOfStrings.isEmpty())
    System.out.println(stackOfStrings.peek());
stackOfStrings.push("Good bye");
for (String two : stackOfStrings)
    System.out.println(two);
```

2. What would be the effect of omitting the conditional test before calling the `peek` method?

3. Would the following statements execute without error? If your answer is "yes," what would their effect be? If "no," why not?

```
stackOfStrings.offer("away");
String three = stackOfStrings.remove();
```

PROGRAMMING

1. Write a fragment that reads a sequence of strings and inserts each string that is not numeric at the front of a deque and each string that is numeric at the rear of a deque. Your fragment should also count the number of strings of each kind.

2. For a deque that has the form required by Programming Exercise 1, display the message `"Strings that are not numeric"` followed by the nonnumeric strings and then the message `"Strings that are numbers"` followed by the numeric strings. Do not empty the deque.

3. Write a `Deque.addFirst` method for class `ArrayQueue`.

Chapter Review

- A stack is a LIFO data structure. This means that the last item added to a stack is the first one removed.
- A stack is a simple but powerful data structure. It has only four operators: empty, peek, pop, and push.
- Stacks are useful when we want to process information in the reverse of the order that it is encountered. For this reason, a stack was used to implement the palindrome finder.
- `java.util.Stack` is implemented as an extension of the `Vector` class. The problem with this approach is that it allows a client to invoke other methods from the deprecated `Vector` class.
- We showed three different ways to implement stacks: using an object of a class that implements the `List` interface as a container; using an array as a container; and using a linked list as a container.
- Stacks can be applied in algorithms for evaluating arithmetic expressions. We showed how to evaluate postfix expressions and how to translate infix expressions with and without parentheses to postfix.
- The queue is an abstract data type with a FIFO structure. This means that the item that has been in the queue the longest will be the first one removed. Queues can be used to represent reservation lists and waiting lines (from which the data structure gets its name "queue").
- The `Queue` interface declares methods `offer`, `remove`, `poll`, `peek`, and `element`.
- We discussed three ways to implement the `Queue` interface: as a double-linked list, as a single-linked list, and as a circular array. All three implementations support insertion and removal in O(1) time; however, there will be a need for reallocation in the circular array implementation (amortized O(1) time). The array implementation requires the smallest amount of storage when it is close to capacity. The `LinkedList` class requires the most storage but no implementation because it is part of `java.util`.
- We discussed the `Deque` interface and showed how its methods allow insertion and removal at either end of a deque. We showed the correspondence between its methods and methods found in the `Stack` class and `Queue` interface.

Java API Classes Introduced in This Chapter

```
java.util.Stack
java.lang.UnsupportedOperationException
java.util.AbstractQueue
java.util.ArrayDeque
java.util.Deque
java.util.NoSuchElementException
java.util.Queue
```

User-Defined Interfaces and Classes in This Chapter

```
ArrayStack
InfixToPostfix
InfixToPostfixParens
IsPalindrome
PalindromeFinder
PostfixEvaluator
StackInt
```

`LinkedStack`
`ListStack`
`ArrayQueue`
`ArrayQueue.Iter`
`SyntaxErrorException`
`ListQueue`
`MaintainQueue`
`KWQueue`

Quick-Check Exercises

1. A stack is a _____ -in, _____ -out data structure.
2. Draw this stack `s` as an object of type `ArrayStack<Character>`. What is the value of data field `topOfStack`?

3. What is the value of `s.empty()` for the stack shown in Question 2?
4. What is returned by `s.pop()` for the stack shown in Question 2?
5. Answer Question 2 for a stack `s` implemented as a linked list (`type LinkedStack<Character>`).
6. Why should the statement `s.remove(i)`, where `s` is of type `StackInt` and `i` is an integer index, not appear in a client program? Can you use this statement with an object of the `Stack` class defined in `java.util`? Can you use it with an object of class `ArrayStack` or `LinkedStack`?
7. What would be the postfix form of the following expression?

   ```
   x + y - 24 * zone - ace / 25 + c1
   ```

   Show the contents of the operator stack just before each operator is processed and just after all tokens are scanned using method `InfixToPostfix.convert`.
8. Answer Question 7 for the following expression.

   ```
   ( x + y - 24 ) * ( zone - ace / ( 25 + c1 ) )
   ```

9. The value of the expression 20 35 – 5 / 10 7 * + is _____. Show the contents of the operand stack just before each operator is processed and just after all tokens are scanned.
10. A queue is a _____ -in, _____ -out data structure.
11. Would a compiler use a stack or a queue in a program that converts infix expressions to postfix?
12. Would an operating system use a stack or a queue to determine which print job should be handled next?
13. Assume that a queue `q` of capacity 6 implemented using a circular array contains the five characters `+, *, -, &, and #` (all wrapped in `Character` objects), where `+` is the first character inserted. Assume that `+` is stored in the first position in the array. What is the value of `q.front`? What is the value of `q.rear`?
14. Remove the first element from the queue in Question 13 and insert the characters \ and %. Draw the new queue. What is the value of `q.front`? What is the value of `q.rear`?
15. If a single-linked list were used to implement the queue in Question 13, the character _____ would be at the head of the list and the character _____ would be at the tail of the list.
16. For a nonempty queue implemented as a single-linked list, the statement _____ would be used inside method `offer` to store a new node whose data field is referenced by `item` in the queue; the statement _____ would be used to disconnect a node after its data was retrieved from the queue.
17. Pick the queue implementation (circular array, single-linked list, double-linked list) that is most appropriate for each of the following conditions.
    a. Storage must be reallocated when the queue is full.
    b. This implementation is normally most efficient in use of storage.
    c. This is an existing class in the Java API.

Review Questions

1. Show the effect of each of the following operations on stack `s`. Assume that `y` (type `Character`) contains the character `'&'`. What are the final values of `x` and `success` and the contents of the stack `s`?

```
Deque<Character> s = new ArrayDeque<>();
char x;
s.push('+');
try {
    x = s.pop();
    success = true;
}
catch (NoSuchElementException e) {
   success = false;
}
try {
    x = s.pop();
    success = true;
}
catch (NoSuchElementException e) {
    success = false;
}
s.push('(');
s.push(y);
try {
     x = s.pop();
     success = true;
}
catch (NoSuchElementException e) {
     success = false;
}
```

2. Write a `toString` method for class `ArrayStack<E>`.
3. Write a `toString` method for class `LinkedStack<E>`.
4. Write an infix expression that would convert to the postfix expression in Quick-Check Question 9.
5. Write a constructor for class `LinkedStack<E>` that loads the stack from an array parameter. The last array element should be at the top of the stack.
6. Write a client that removes all negative numbers from a stack of `Integer` objects. If the original stack contained the integers 30, –15, 20, –25 (top of stack), the new stack should contain the integers 30, 20.
7. Write a method `peekNextToTop` that allows you to retrieve the element just below the one at the top of the stack without removing it. Write this method for both `ArrayStack<E>` and `LinkedStack<E>`. It should return `null` if the stack has just one element, and it should throw an exception if the stack is empty.
8. Show the effect of each of the following operations on queue `q`. Assume that `y` (type `Character`) contains the character `'&'`. What are the final values of `x` and `success` (type `boolean`) and the contents of queue `q`?

```
Queue<Character> q = new ArrayQueue<>();
boolean success = true;
char x;
q.offer('+');
try {
    x = q.remove();
    x = q.remove();
    success = true;
} catch(NoSuchElementException e) {
    success = false;
}
```

```
q.offer('(');
q.offer(y);
try {
    x = q.remove(); success = true;
} catch(NoSuchElementException e) {
    success = false;
}
```

9. Write a new queue method called `moveToRear` that moves the element currently at the front of the queue to the rear of the queue. The element that was second in line will be the new front element. Do this using methods `Queue.offer` and `Queue.remove`.
10. Answer Question 9 without using methods `Queue.offer` or `Queue.remove` for a single-linked list implementation of Queue. You will need to manipulate the queue's internal data fields directly.
11. Answer Question 9 without using methods `Queue.offer` or `Queue.remove` for a circular array implementation of `Queue`. You will need to manipulate the queue internal data fields directly.
12. Write a new queue method called `moveToFront` that moves the element at the rear of the queue to the front of the queue, while the other queue elements maintain their relative positions behind the old front element. Do this using methods `Queue.offer` and `Queue.remove`.
13. Answer Question 12 without using `Queue.offer` and `Queue.remove` for a single-linked list implementation of Queue.
14. Answer Question 12 without using methods `Queue.offer` or `Queue.remove` for a circular array implementation of Queue.

Programming Projects

1. Add a method `isPalindromeLettersOnly` to the `PalindromeFinder` class that bases its findings only on the letters in a string (ignoring spaces, digits, and other characters that are not letters).

2. Develop an Expression Manager that can do the following operations:

   Balanced Symbols Check
   - Read a mathematical expression from the user.
   - Check and report whether the expression is balanced.
   - `'{','}','(',')','[',']'` are the only characters considered for the check. All other characters can be ignored.

   Infix to Postfix Conversion
   - Read an infix expression from the user.
   - Perform the Balanced Parentheses Check on the expression read.
   - If the expression fails the Balanced Parentheses Check, report a message to the user that the expression is invalid.
   - If the expression passes the Balanced Parentheses Check, convert the infix expression into a postfix expression and display it to the user.
   - Operators to be considered are `+, -, *, /, %`.

   Postfix to Infix Conversion
   - Read a postfix expression from the user.
   - Convert the postfix expression into an infix expression and display it to the user.
   - Display an appropriate message if the postfix expression is not valid.
   - Operators to be considered are `+, -, *, /, %`.

   Evaluating a Postfix Expression
   - Read the postfix expression from the user.
   - Evaluate the postfix expression and display the result.

- Display an appropriate message if the postfix expression is not valid.
- Operators to be considered are +, -, *, /, %.
- Operands should be only integers.

Implementation

- Design a menu that has buttons or requests user input to select from all the aforementioned operations.

3. Write a program to handle the flow of widgets into and out of a warehouse. The warehouse will have numerous deliveries of new widgets and orders for widgets. The widgets in a filled order are billed at a profit of 50 percent over their cost. Each delivery of new widgets may have a different cost associated with it. The accountants for the firm have instituted a LIFO system for filling orders. This means that the newest widgets are the first ones sent out to fill an order. Also, the most recent orders are filled first. This method of inventory can be represented using two stacks: orders-to-be-filled and widgets-on-hand. When a delivery of new widgets is received, any unfilled orders (on the orders-to-be-filled stack) are processed and filled. After all orders are filled, if there are widgets remaining in the new delivery, a new element is pushed onto the widgets-on-hand stack. When an order for new widgets is received, one or more objects are popped from the widgets-on-hand stack until the order has been filled. If the order is completely filled and there are widgets left over in the last object popped, a modified object with the quantity updated is pushed onto the widgets-on-hand stack. If the order is not completely filled, the order is pushed onto the orders-to-be-filled stack with an updated quantity of widgets to be sent out later. If an order is completely filled, it is not pushed onto the stack.

   Write a class with methods to process the shipments received and to process orders. After an order is filled, display the quantity sent out and the total cost for all widgets in the order. Also indicate whether there are any widgets remaining to be sent out at a later time. After a delivery is processed, display information about each order that was filled with this delivery and indicate how many widgets, if any, were stored in the object pushed onto the widgets-on-hand stack.

4. You can combine the algorithms for converting between infix to postfix and for evaluating postfix to evaluate an infix expression directly. To do so you need two stacks: one to contain operators and the other to contain operands. When an operand is encountered, it is pushed onto the operand stack. When an operator is encountered, it is processed as described in the infix to postfix algorithm. When an operator is popped off the operator stack, it is processed as described in the postfix evaluation algorithm: The top two operands are popped off the operand stack, the operation is performed, and the result is pushed back onto the operand stack. Write a program to evaluate infix expressions directly using this combined algorithm.

5. Write a client program that uses the `Stack` abstract data type to compile a simple arithmetic expression without parentheses. For example, the expression

   ```
   a + b * c - d
   ```

   should be compiled according to the following table:

Operator	Operand 1	Operand 2	Result
*	b	c	z
+	a	z	y
-	y	d	x

   The table shows the order in which the operations are performed (*, +, -) and operands for each operator. The result column gives the name of an identifier (working backward from `z`) chosen to hold each result. Assume the operands are the letters a through `m` and the operators are (+, -, *, /). Your program should read each character and process it as follows: If the character is blank, ignore it. If the character is neither blank nor an operand nor an operator, display an error message and terminate the program. If it is an operand, push it onto the operand stack. If it is an operator,

compare its precedence to that of the operator on top of the operator stack. If the current operator has higher precedence than the one currently on top of the stack (or stack is empty), it should be pushed onto the operator stack. If the current operator has the same or lower precedence, the operator on top of the operator stack must be evaluated next. This is done by popping that operator off the operator stack along with a pair of operands from the operand stack and writing a new line in the output table. The character selected to hold the result should then be pushed onto the operand stack. Next, the current operator should be compared to the new top of the operator stack. Continue to generate output lines until the top of the operator stack has lower precedence than the current operator or until it is empty. At this point, push the current operator onto the top of the stack and examine the next character in the data string. When the end of the string is reached, pop any remaining operator along with its operand pair just described. Remember to push the result character onto the operand stack after each table line is generated.

6. Another approach to checking for palindromes would be to store the characters of the string being checked in a stack and then remove half of the characters, pushing them onto a second stack. When you are finished, if the two stacks are equal, then the string is a palindrome. This works fine if the string has an even number of characters. If the string has an odd number of characters, an additional character should be removed from the original stack before the two stacks are compared. It doesn't matter what this character is because it doesn't have to be matched. Design, code, and test a program that implements this approach.

7. Operating systems sometimes use a fixed array storage area to accommodate a pair of stacks such that one grows from the bottom (with its first item stored at index 0) and the other grows from the top (with its first item stored at the highest array index). As the stacks grow, the top of the stacks will move closer together.

   The stacks are full when the two top elements are stored in adjacent array elements (`top2 == top1 + 1`). Design, code, and test a class `DoubleStack` that implements this data structure. `DoubleStack` should support the normal stack operations (`push, pop, peek, empty`, etc.). Each stack method should have an additional `int` parameter that indicates which of the stacks (1 or 2) is being processed. For example, `push(1, item)` will push `item` onto stack 1.

8. Redo Programming Project 3, assuming that widgets are shipped using a FIFO inventory system.

9. Write a class `MyArrayDeque` that extends class `ArrayQueue`. Class `MyArrayDeque` should implement the `Deque` interface. Test your new class by comparing its operation to that of the `ArrayDeque` class in the Java Collections Framework.

10. Write a program that reads in a sequence of characters and stores each character in a deque. Display the deque contents. Then use a second deque to store the characters in reverse order. When done, display the contents of both deques.

11. An operating system assigns jobs to print queues based on the number of pages to be printed (less than 10 pages, less than 20 pages, or more than 20 pages but less than 50 pages). You may assume that the system printers are able to print 10 pages per minute. Smaller print jobs are printed before larger print jobs, and print jobs of the same priority are queued up in the order in which they are received. The system administrators would like to compare the time required to process a set of print jobs using one, two, or three system printers.

    Write a program that simulates processing 100 print jobs of varying lengths using one, two, or three printers. Assume that a print request is made every minute and that the number of pages to print varies from 1 to 50 pages.

    The output from your program should indicate the order in which the jobs were received, the order in which they were printed, and the time required to process the set of print jobs. If more than one printer is being used, indicate which printer each job was printed on.

12. Write a menu-driven program that uses an array of queues to keep track of a group of executives as they are transferred from one department to another, get paid, or become unemployed. Executives within a department are paid based on their seniority, with the person who has been in the department the longest receiving the most money. Each person in the department receives $1000 in salary for each person in her department having less seniority than she has. Persons who are unemployed receive no compensation.

    Your program should be able to process the following set of commands:

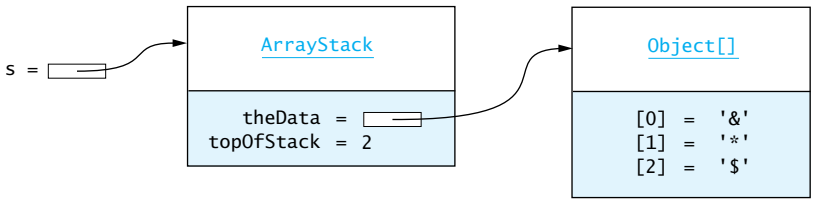
Join <*person*> <*department*>	<*person*> is added to <*department*>
Quit <*person*>	<*person*> is removed from his or her department
Change <*person*> <*department*>	<*person*> is moved from old department to <*department*>
Payroll	Each executive's salary is computed and displayed by department in decreasing order of seniority

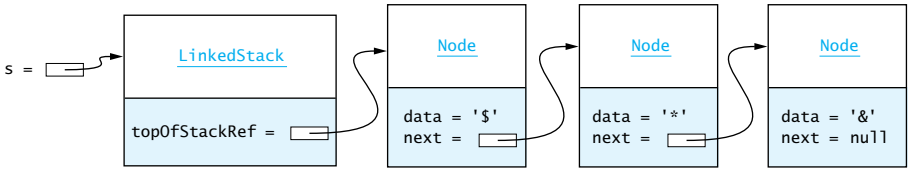
    Hint: You might want to include a table that contains each executive's name and information and the location of the queue that contains his or her name, to make searching more efficient.

13. A *randomized queue* is similar to a queue, except that the item removed is chosen at random from the items in the queue. Create a `RandomizedQueue` that contains the normal `queue` methods except that the `remove` method will delete an item chosen using a uniform distribution. You should write this class as an extension of the `ArrayQueue` class.

Answers to Quick-Check Exercises

1. A stack is a LIFO data structure.
2. Each character in array `theData` should be wrapped in a `Character` object. The value of `topOfStack` should be 2.



3. Method `empty` returns `false`.
4. `pop` returns a reference to the `Character` object that wraps `'$'`.
5. 

6. Method `remove(int i)` is not defined for classes that implement interface `StackInt`. The `Stack` class defined in API `java.util` would permit its use. Classes `ArrayStack` and `LinkedStack` would not.
7. Infix: `x + y - 24 * zone - ace / 25 + cl`

    Postfix: `x y + 24 zone * - ace 25 / - cl +`

    Operator stack before first `+` : | Empty stack (vertical bar is bottom of stack)

    Operator stack before first `-` : | `+`

Operator stack before first * : |-
Operator stack before second – : | -, *
Operator stack before first / : | -
Operator stack before second + : | -,/
Operator stack after all tokens scanned: | +

8 Infix: `( x + y - 24 ) * ( zone - ace / ( 25 + cl ) )`

Postfix: `x y + 24 - zone ace 25 cl + / - *`

Operator stack before first (: | Empty stack (vertical bar is bottom of stack)
Operator stack before first + : | (
Operator stack before first – : | (,+
Operator stack before first) : | (,-;
Operator stack before first * : | Empty stack
Operator stack before second (: | *
Operator stack before second – : |*, (
Operator stack before second / : | *,(,-
Operator stack before third (: | *,(,-,/
Operator stack before second + : | *,(,-, /, (
Operator stack before second) : | *,(,-, /, (, +
Operator stack before third) : | *,(,-, /
Operator stack after all tokens scanned: | *

9. `20 35 - 5 / 10 7 * + is 67`

Operand stack just before - : | 20, 35
Operand stack just before/ : | -15, 5
Operand stack just before * : | -3, 10, 7
Operand stack just before + : | -3, 70
Operand stack after all tokens scanned: | 67

10. *first*, *first*

11. `stack`

12. `queue`

13. `q.front is 0; q.rear is 4.`

14. `q.rear` `q.front`

%
*
-
&
#
\

`q.front is 1; q.rear is 0`

15. `'*' '%'`

16. For insertion: `rear.next = new Node<E>(item);`
To disconnect the node removed: `front = front.next;`

17. a. circular array
b. single-linked list
c. double-linked list (`class LinkedList`)

Chapter 5

Recursion

Chapter Objectives

- To understand how to think recursively
- To learn how to trace a recursive method
- To learn how to write recursive algorithms and methods for searching arrays
- To learn about recursive data structures and recursive methods for a `LinkedList` class
- To understand how to use recursion to solve the Towers of Hanoi problem
- To understand how to use recursion to process two-dimensional images
- To learn how to apply backtracking to solve search problems such as finding a path through a maze

This chapter introduces a programming technique called recursion and shows you how to think recursively. You can use recursion to solve many kinds of programming problems that would be very difficult to conceptualize and solve without recursion. Computer scientists in the field of artificial intelligence (AI) often use recursion to write programs that exhibit intelligent behavior: playing games such as chess, proving mathematical theorems, recognizing patterns, and so on.

In the beginning of the chapter, you will be introduced to recursive thinking and how to design a recursive algorithm and prove that it is correct. You will also learn how to trace a recursive method and use activation frames for this purpose.

Recursive algorithms and methods can be used to perform common mathematical operations, such as computing a factorial or a greatest common divisor (gcd). Recursion can be used to process familiar data structures, such as strings, arrays, and linked lists, and to design a very efficient array search technique called binary search. You will also see that a linked list is a recursive data structure and learn how to write recursive methods that perform common list-processing tasks.

Recursion can be used to solve a variety of other problems. The case studies in this chapter use recursion to solve a game, to search for "blobs" in a two-dimensional image, and to find a path through a maze.

Recursion

5.1 Recursive Thinking

Recursion is a problem-solving approach that can be used to generate simple solutions to certain kinds of problems that would be difficult to solve in other ways. In a recursive algorithm, the original problem is split into one or more simpler versions of itself. For example, if the solution to the original problem involved n items, recursive thinking might split it into two problems: one involving $n - 1$ items and one involving just a single item. Then the problem with $n - 1$ items could be split again into one involving $n - 2$ items and one involving just a single item, and so on. If the solution to all the one-item problems is "trivial," we can build up the solution to the original problem from the solutions to the simpler problems.

As an example of how this might work, consider a collection of nested wooden figures as shown in Figure 5.1. If you wanted to write an algorithm to "process" this collection in some way (such as counting the figures or painting a face on each figure), you would have difficulty doing it because you don't know how many objects are in the nest. But you could use recursion to solve the problem in the following way.

Recursive Algorithm to Process Nested Figures

1. `if` there is one figure in the nest
2. Do whatever is required to the figure.

   `else`
3. Do whatever is required to the outer figure in the nest.
4. Process the nest of figures inside the outer figure in the same way.

FIGURE 5.1 A Set of Nested Wooden Figures

In this recursive algorithm, the solution is trivial if there is only one figure: perform Step 2. If there is more than one figure, perform Step 3 to process the outer figure. Step 4 is the recursive operation—recursively process the nest of figures inside the outer figure. This nest will, of course, have one less figure than before, so it is a simpler version of the original problem.

As another example, let's consider searching for a target value in an array. Assume that the array elements are sorted and are in increasing order. A recursive approach, which we will study in detail in Section 5.3, involves replacing the problem of searching an array of n elements with one of searching an array of $n/2$ elements. How do we do that? We compare the target value to the value of the element in the middle of the sorted array. If there is a match, we have found the target. If not, based on the result of the comparison, we either search the elements that come before the middle one or the elements that come after the middle one. So we have replaced the problem of searching an array with n elements to one that involves searching a smaller array with only $n/2$ elements. The recursive algorithm follows.

Recursive Algorithm to Search an Array

1. `if` the array is empty
2. Return –1 as the search result.

   `else if` the middle element matches the target
3. Return the subscript of the middle element as the result.

   `else if` the target is less than the middle element
4. Recursively search the array elements before the middle element and return the result.

   `else`
5. Recursively search the array elements after the middle element and return the result.

The condition in Step 1 is true when there are no elements left to search. Step 2 returns –1 to indicate that the search failed. Step 3 executes when the middle element matches the target. Otherwise, we recursively apply the search algorithm (Steps 4 and 5), thereby searching a smaller array (approximately half the size), and return the result. For each recursive search, the region of the array being searched will be different, so the middle element will also be different.

The two recursive algorithms we showed so far follow this general approach:

General Recursive Algorithm

1. `if` the problem can be solved for the current value of n
2. Solve it.

   `else`
3. Recursively apply the algorithm to one or more problems involving smaller values of n.
4. Combine the solutions to the smaller problems to get the solution to the original.

Step 1 involves a test for what is called the *base case*: the value of n for which the problem can be solved easily. Step 3 is the *recursive case* because we recursively apply the algorithm. Because the value of n for each recursive case is smaller than the original value of n, each recursive case makes progress toward the base case. Whenever a split occurs, we revisit Step 1 for each new problem to see whether it is a base case or a recursive case.

Steps to Design a Recursive Algorithm

From what we have seen so far, we can summarize the characteristics of a recursive solution:

- There must be at least one case (the base case), for a small value of *n*, that can be solved directly.
- A problem of a given size (say, *n*) can be split into one or more smaller versions of the same problem (the recursive case).

Therefore, to design a recursive algorithm, we must

- Recognize the base case and provide a solution to it.
- Devise a strategy to split the problem into smaller versions of itself. Each recursive case must make progress toward the base case.
- Combine the solutions to the smaller problems in such a way that each larger problem is solved correctly.

Next, we look at a recursive algorithm for a common programming problem. We will also provide a Java method that solves this problem. All of the methods in this section and in the next will be found in class `RecursiveMethods.java` on this textbook's Web site.

EXAMPLE 5.1 Let's see how we could write our own recursive method for finding the string length. How would you go about doing this? If there is a special character that marks the end of a string, then you can count all the characters that precede this special character. But if there is no special character, you might try a recursive approach. The base case is an empty string—its length is 0. For the recursive case, consider that each string has two parts: the first character and the "rest of the string." If you can find the length of the "rest of the string," you can then add 1 (for the first character) to get the length of the larger string. For example, the length of "abcde" is 1 + the length of "bcde".

Recursive Algorithm for Finding the Length of a String

1. `if` the string is empty (has no characters)
2. The length is 0.

   `else`
3. The length is 1 plus the length of the string that excludes the first character.

We can implement this algorithm as a `static` method with a `String` argument. The test for the base case is a string reference of `null` or a string that contains no characters (`""`). In either case, the length is 0. In the recursive case,

```
return 1 + length(str.substring(1));
```

the method `call str.substring(1)` returns a reference to a string containing all characters in `string str` except for the character at position 0. Then we call method `length` again with this substring as its argument. The method result is one more than the value returned from the next call to `length`. Each time we reenter the method `length`, the `if` statement executes with `str` referencing a string containing all but the first character in the previous call. Method `length` is called a *recursive method* because it calls itself.

```
/** Recursive method length (in RecursiveMethods.java).
    @param str The string
    @return The length of the string
 */
```

```
public static int length(String str) {
    if (str == null || str.isEmpty()) {
        return 0;
    } else {
        return 1 + length(str.substring(1));
    }
}
```

EXAMPLE 5.2 Method `printChars` is a recursive method that displays each character in its string argument on a separate line. In the base case (an empty or nonexistent string), the method return occurs immediately and nothing is displayed. In the recursive case, `printChars` displays the first character of its string argument and then calls itself to display the characters in the rest of the string. If the initial call is `printChars("hat")`, the method will display the lines

```
h
a
t
```

Unlike the method `length` in Example 5.1, `printChars` is a `void` method. However, both methods follow the format for the general recursive algorithm shown earlier.

```
/** Recursive method printChars (in RecursiveMethods.java).
    post: The argument string is displayed, one character per line.
    @param str The string
 */
public static void printChars(String str) {
    if (str == null || str.isEmpty()) {
        return;
    } else {
        System.out.println(str.charAt(0));
        printChars(str.substring(1));
    }
}
```

You get an interesting result if you reverse the two statements in the recursive case.

```
/** Recursive method printCharsReverse (in RecursiveMethods.java).
    post: The argument string is displayed in reverse, one character per line.
    @param str The string
 */
public static void printCharsReverse(String str) {
    if (str == null || str.isEmpty()) {
        return;
    } else {
        printCharsReverse(str.substring(1));
        System.out.println(str.charAt(0));
    }
}
```

Method `printCharsReverse` calls itself to display the rest of the string before displaying the first character in the current string argument. The effect will be to delay displaying the first character in the current string until all characters in the rest of the string are displayed. Consequently, the characters in the string will be displayed in reverse order. If the initial call is `printCharsReverse("hat")`, the method will display the lines

```
t
a
h
```

Proving that a Recursive Method Is Correct

To prove that a recursive method is correct, you must verify that you have performed correctly the design steps listed earlier. You can use a technique that mathematicians use to prove that a theorem is true for all values of n. A *proof by induction* works the following way:

- Prove the theorem is true for the base case (usually $n = 0$ or $n = 1$).
- Show that if the theorem is assumed true for n, then it must be true for $n + 1$.

We can extend the notion of an inductive proof and use it as the basis for proving that a recursive algorithm is correct. To do this:

- Verify that the base case is recognized and solved correctly.
- Verify that each recursive case makes progress toward the base case.
- Verify that if all smaller problems are solved correctly, then the original problem is also solved correctly.

If you can show that your algorithm satisfies these three requirements, then your algorithm will be correct.

EXAMPLE 5.3 To prove that the `length` method is correct, we know that the base case is an empty string and its length is correctly set at 0. The recursive case involves a call to `length` with a smaller string, so it is making progress toward the base case. Finally, if we know the length of the rest of the string, adding 1 gives us the length of the longer string consisting of the first character and the rest of the string.

Tracing a Recursive Method

Figure 5.2 traces the execution of the method call `length("ace")`. The diagram shows a sequence of recursive calls to the method `length`. After returning from each call to `length`, we complete execution of the statement `return 1 + length(...);` by adding 1 to the result so far and then returning from the current call. The final result, 3, would be returned from the original call. The arrow alongside each word `return` shows which call to `length` is associated with that result. For example, 0 is the result of the method call `length("")`. After adding 1, we return 1, which is the result of the call `length("e")`, and so on. This process of returning from the recursive calls and computing the partial results is called *unwinding the recursion*.

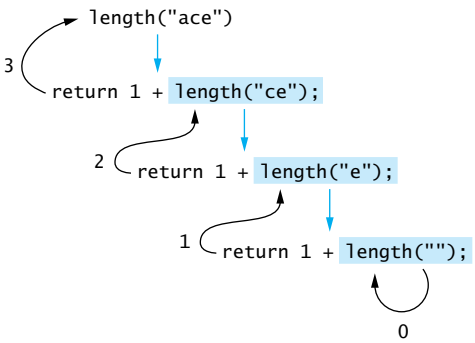


FIGURE 5.2 Trace of `length("ace")`

The Run-Time Stack and Activation Frames

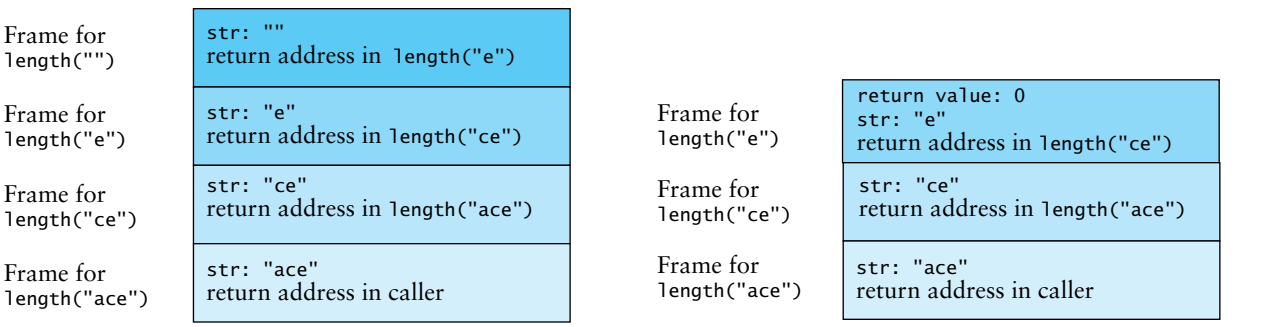
You can also trace a recursive method by showing what Java does when one method calls another. Java maintains a run-time stack, on which it saves new information in the form of an *activation frame*. The activation frame contains storage for the method arguments and any local variables as well as the return address of the instruction that called the method. Whenever a method is called, Java pushes a new activation frame onto the run-time stack and saves this information on the stack. This is done whether or not the method is recursive.

The left side of Figure 5.3 shows the activation frames on the run-time stack after the last recursive call (corresponding to `length("")`) resulting from an initial call to `length("ace")`. At any given time, only the frame at the top of the stack is accessible, so its argument values will be used when the method instructions execute. When the return statement executes, control will be passed to the instruction at the specified return address, and the frame at the top will be popped from the stack (Figure 5.3, right). The activation frame corresponding to the next-to-last call (`length("e")`) is now accessible.

You can think of the run-time stack for a sequence of calls to a recursive method as an office tower in which an employee on each floor has the same list of instructions.[1] The employee in the bottom office carries out part of the instructions on the list, calls the employee in the office above, and is put on hold. The employee in the office above starts to carry out the list of instructions, calls the employee in the next higher office, is put on hold, and so on. When the employee on the top floor is called, that employee carries out the list of instructions to completion and then returns an answer to the employee below. The employee below then resumes carrying out the list of instructions and returns an answer to the employee on the next lower floor, and so on, until an answer is returned to the employee in the bottom office, who then resumes carrying out the list of instructions.

To make the flow of control easier to visualize, we will draw the activation frames from the top of the page down (see Figure 5.4). For example, the activation frame at the top, which would actually be at the bottom of the run-time stack, represents the first call to the recursive method. The downward-pointing arrows connect each statement that calls a method with the frame for that particular execution of the method. The upward-pointing arrows show the

FIGURE 5.3
Run-Time Stack before and after Removal of Frame for `length("")`



Run-time stack after all calls

Run-time stack after return from `length("")`

[1] Analogy suggested by Richard Pattis, University of California, Irvine, CA.

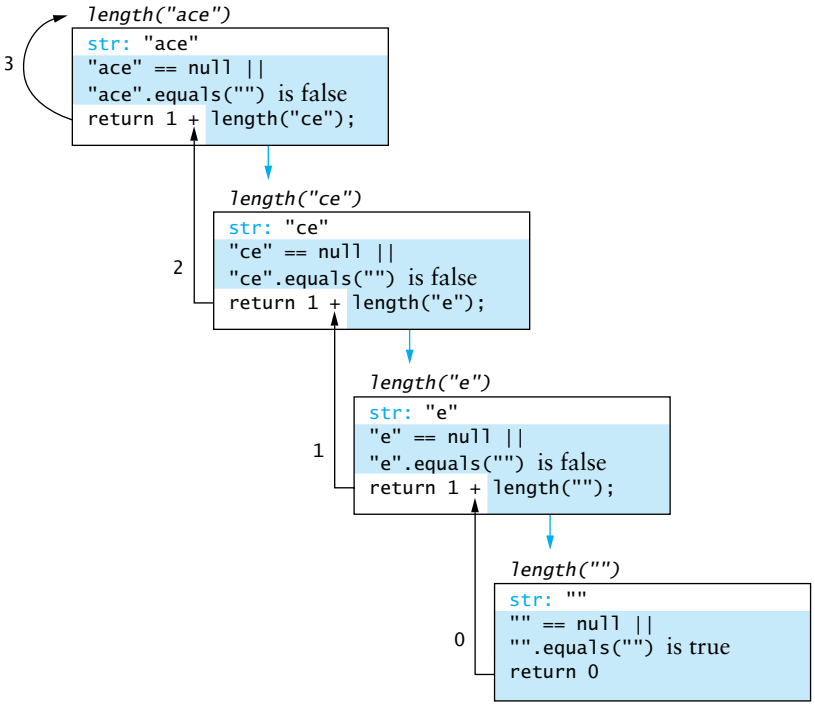


FIGURE 5.4 Trace of `length("ace")` Using Activation Frames

return point from each lower-level call with the value returned alongside the arrow. For each frame, the return point is to the addition operator in the statement `return 1 + length(...);`. For each frame, the code in the blue screen is executed prior to the creation of the next activation frame; the rest of the code shown is executed after the return.

EXERCISES FOR SECTION 5.1

SELF-CHECK

1. Trace the execution of the `call mystery(4)` for the following recursive method using activation frames as shown in Figure 5.4. What does this method do?

```
public static mystery(int n) {
    if (n == 0)
        return 0;
    else
        return n * n + mystery(n - 1);
}
```

2. A recursive method `upperCount` that counts the number of uppercase letters in a string would implement the following algorithm:

   if the string length is 0 **return** 0
   else if the last character is uppercase return 1 + `upperCount` for the rest of the string
   else return `upperCount` for the rest of the string.

   Trace its execution using activation frames on the string `"John Doe"`.

3. Trace the execution of method `toNumber` described in Programming Exercise 1 below using activation frames.
4. Trace the execution of `printCharsReverse("toc")` using activation frames.
5. Prove that the `printChars` method is correct.
6. Trace the execution of `length("tictac")` using a diagram like Figure 5.2.
7. Write a recursive algorithm that determines whether a specified target character is present in a string. It should return `true` if the target is present and `false` if it is not. The stopping steps should be
   a. a string reference to null or a string of length 0, the result is false
   b. the first character in the string is the target, the result is true.

   The recursive step would involve searching the rest of the string.

PROGRAMMING

1. Write a recursive method `toNumber` that forms the integer sum of all digit characters in a string. For example, the result of `toNumber("3ac4")` would be 7. *Hint*: If `next` is a digit character (`'0' through '9'`), `Character.isDigit(next)` is true and the numeric value of `next` is `Character.digit(next, 10)`.
2. Write a recursive method `double` that returns a string with each character in its argument repeated. For example, if the string passed to `double` is `"hello"`, `double` will return the string `"hheellllloo"`.
3. Write a recursive method that implements the recursive algorithm for searching a string in Self-Check Exercise 7. The method heading should be

   ```
   public static boolean searchString(String str, char ch)
   ```

4. Write the method described in Self-Check Exercise 2.

5.2 Recursive Definitions of Mathematical Formulas

Mathematicians often use recursive definitions of formulas. These definitions lead very naturally to recursive algorithms.

EXAMPLE 5.4 The factorial of n, or $n!$, is defined as follows:

$$0! = 1$$
$$n! = n \times (n-1)!$$

The first formula identifies the base case: n equal to 0. The second formula is a recursive definition. It leads to the following algorithm for computing $n!$.

Recursive Algorithm for Computing *n*!

1. `if` n equals 0
2. $\quad n!$ is 1.

   `else`
3. $\quad n! = n \times (n - 1)!$

To verify the correctness of this algorithm, we see that the base case is solved correctly (0! is 1). The recursive case makes progress toward the base case because it involves the calculation of a smaller factorial. Also, if we can calculate $(n - 1)!$, the recursive case gives us the correct formula for calculating $n!$.

The recursive method follows. The statement

```
return n * factorial(n - 1);
```

implements the recursive case. Each time `factorial` calls itself, the method body executes again with a different argument value. An initial method call such as `factorial(4)` will generate four recursive calls, as shown in Figure 5.5.

```
/** Recursive factorial method (in RecursiveMethods.java).
    pre: n >= 0
    @param n The integer whose factorial is being computed
    @return n!
 */
public static int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

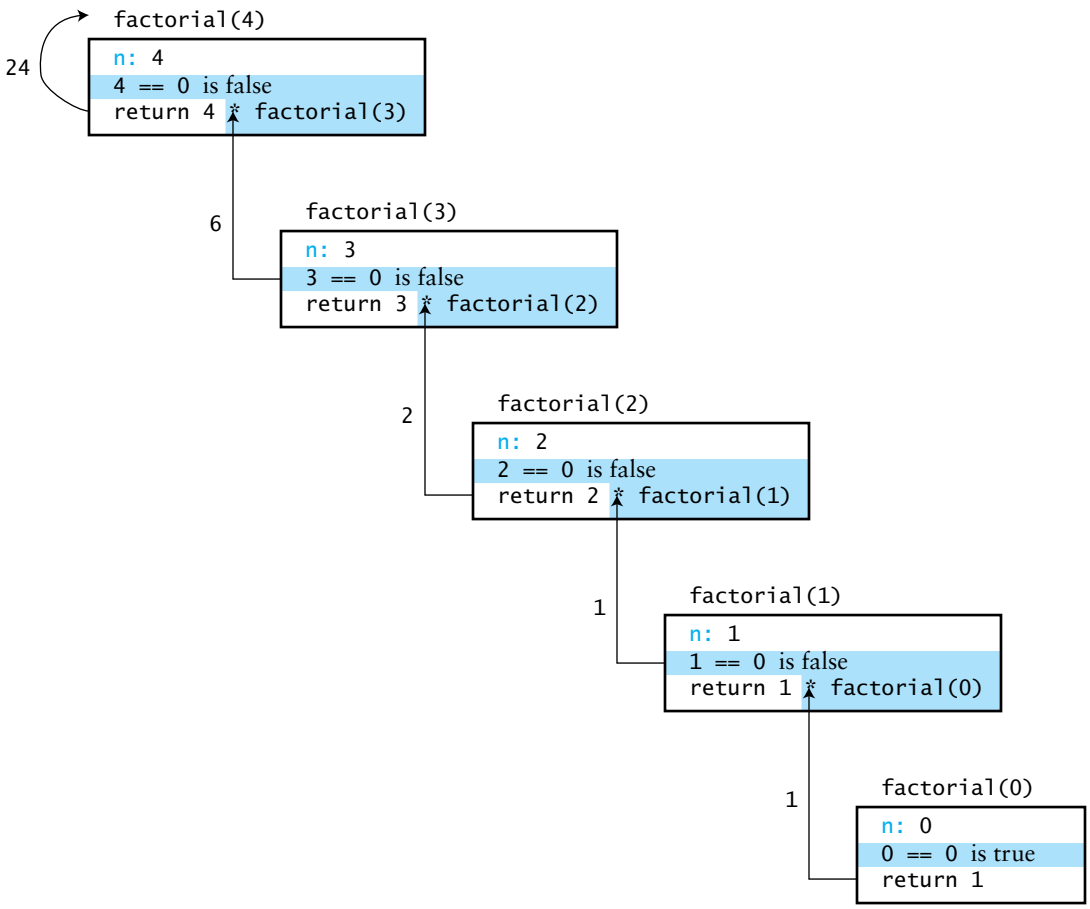


FIGURE 5.5 Trace of `factorial(4)`

PITFALL

Infinite Recursion and Stack Overflow

If you call method factorial with a negative argument, you will see that the recursion does not terminate. It will continue forever because the stopping case, *n* equals 0, can never be reached, as *n* gets more negative with each call. For example, if the original value of *n* is -4, you will make method calls `factorial(-5)`, `factorial(-6)`, `factoria1(-7)`, and so on. You should make sure that your recursive methods are constructed so that a stopping case is always reached. One way to prevent the infinite recursion in this case would be to change the terminating condition to *n* <= 0.

However, this would incorrectly return a value of 1 for *n*! if *n* is negative. A better solution would be to throw an `IllegalArgumentException` if *n* is negative.

If your program does not terminate properly, you may see an extremely long display on the console (if the console is being used to display its results). Eventually the exception `StackOverflowError` will be thrown. This means that the memory area used to store information about method calls (the run-time stack) has been used up because there have been too many calls to the recursive method. Because there is no memory available for this purpose, your program can't execute any more method calls.

EXAMPLE 5.5 Let's develop a recursive method that raises a number x to a power n, where n is positive or zero. You can raise a number to a power by repeatedly multiplying that number by itself. So if we know x^k, we can get x^{k+1} by multiplying x^k by x. The recursive definition is

$$x^n = x \times x^{n-1}$$

This gives us the recursive case. You should know that any number raised to the power 0 is 1, so the base case is

$$x^0 = 1$$

Recursive Algorithm for Calculating x^n ($n \geq 0$)

1. **if** *n* is 0
2. The result is 1.

else

3. The result is $x \times x^{n-1}$.

We show the method next.

```
/** Recursive power method (in RecursiveMethods.java).
    pre: n >= 0
    @param x The number being raised to a power
    @param n The exponent
    @return x raised to the power n
 */
```

```
public static double power(double x, int n) {
    if (n == 0)
        return 1;
    else
        return x * power(x, n - 1);
}
```

EXAMPLE 5.6 The greatest common divisor (gcd) of two numbers is the largest integer that divides both numbers. For example, the gcd of 20, 15 is 5; the gcd of 36, 24 is 12; the gcd of 36, 18 is 18. The mathematician Euclid devised an algorithm for finding the greatest common divisor (gcd) of two integers, *m* and *n*, based on the following definition.

Definition of gcd(*m*, *n*) for *m* > *n*

gcd(*m*, *n*) = *n* if *n* is a divisor of *m*
gcd(*m*, *n*) = gcd(*n*, *m* % *n*) if *n* isn't a divisor of *m*

This definition states that gcd(*m*, *n*) is *n* if *n* divides *m*. This is correct because no number larger than *n* can divide *n*. Otherwise, the definition states that gcd(*m*, *n*) is the same as gcd(*n*, *m* % *n*), where *m* % *n* is the integer remainder of *m* divided by *n*. Therefore, gcd(20, 15) is the same as gcd(15, 5), or 5, because 5 divides 15. This recursive definition leads naturally to a recursive algorithm.

Recursive Algorithm for Calculating gcd(*m*, *n*) for *m* > *n*

1. **if** *n* is a divisor of *m*
2. The result is *n*.

else

3. The result is gcd (*n*, *m* % *n*).

To verify that this is correct, we need to make sure that there is a base case and that it is solved correctly. The base case is "*n* is a divisor of *m*." If so, the solution is *n* (*n* is the gcd), which is correct. Does the recursive case make progress to the base case? It must because both arguments in each recursive call are smaller than in the previous call, and the new second argument is always smaller than the new first argument (*m* % *n* must be less than *n*). Eventually a divisor will be found, or the second argument will become 1. Since 1 is a base case (1 divides every integer), we have verified that the recursive case makes progress toward the base case.

Next, we show method `gcd`. Note that we do not need a separate clause to handle arguments that initially are not in the correct sequence. This is because if `m < n`, then `m % n` is `m` and the recursive call will transpose the arguments so that `m > n` in the first recursive call.

```
/** Recursive gcd method (in RecursiveMethods.java).
    pre: m > 0 and n > 0
    @param m The larger number
    @param n The smaller number
    @return Greatest common divisor of m and n
 */
public static double gcd(int m, int n) {
    if (m % n == 0)
        return n;
    else
        return gcd(n, m % n);
}
```

Tail Recursion versus Iteration

Method gcd above is an example of *tail recursion*. In tail recursion, the last thing a method does is to call itself. You may have noticed that there are some similarities between tail recursion and iteration. Both techniques enable us to repeat a compound statement. In iteration, a loop repetition condition in the loop header determines whether we repeat the loop body or exit from the loop. We repeat the loop body while the repetition condition is true. In tail recursion, the condition usually tests for a base case. In a recursive method, we stop the recursion when the base case is reached (the condition is true), and we execute the method body again when the condition is false.

We can always write an iterative solution to any problem that is solvable by recursion. However, the recursive solutions will be easier to conceptualize and should, therefore, lead to methods that are easier to write, read, and debug—all of which are very desirable attributes of code.

EXAMPLE 5.7 In Example 5.4, we wrote the recursive method.

```
public static int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

It is a straightforward process to turn this method into an iterative one, replacing the **if** statement with a loop, as we show next.

```
/** Iterative factorial method.
    pre: n >= 0
    @param n The integer whose factorial is being computed
    @return n!
 */
public static int factorial1Iter(int n) {
    int result = 1;
    while (n > 0){
        result *= n;
        n = n - 1;
    }
    return result;
}
```

Efficiency of Recursion

The iterative method `factorialIter` multiplies all integers between 1 and n to compute $n!$. It may be slightly less readable than the recursive method `factorial`, but not much. In terms of efficiency, both algorithms are $O(n)$ because the number of loop repetitions or recursive calls increases linearly with n. However, the iterative version is probably faster because the overhead for a method call and return would be greater than the overhead for loop repetition (testing and incrementing the loop control variable). The difference, though, would not be significant. Generally, if it is easier to conceptualize an algorithm using recursion, then you should code it as a recursive method because the reduction in efficiency does not outweigh the advantage of readable code that is easy to debug.

EXAMPLE 5.8 The Fibonacci numbers fib_n are a sequence of numbers that were invented to model the growth of a rabbit colony. Therefore, we would expect this sequence to grow very quickly, and it does. For example, fib_{10} is 55, fib_{15} is 610, fib_{20} is 6765, and fib_{25} is 75,025 (that's a lot of rabbits!). The definition of this sequence follows:

$fib_0 = 0$
$fib_1 = 1$
$fib_n = fib_{n-1} + fib_{n-2}$

Next, we show a method that calculates the *n*th Fibonacci number. The last line codes the recursive case.

```
/** Recursive method to calculate Fibonacci numbers
    (in RecursiveMethods.java).
    pre: n >= 0
    @param n The position of the Fibonacci number being calculated
    @return The Fibonacci number
 */
public static int fibonacci(int n) {
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Unfortunately, this solution is very inefficient because of multiple calls to `fibonacci` with the same argument. For example, calculating `fibonacci(5)` results in calls to `fibonacci(4)` and `fibonacci(3)`. Calculating `fibonacci(4)` results in calls to `fibonacci(3)` (second call) and also `fibonacci(2)`. Calculating `fibonacci(3)` twice results in two more calls to `fibonacci(2)` (three calls total), and so on (see Figure 5.6).

Because of the redundant method calls, the time required to calculate `fibonacci(n)` increases exponentially with `n`. For example, if `n` is 100, there are approximately 2^{100} activation frames. This number is approximately 10^{30}. If you could process one million activation frames per second, it would still take 10^{24} seconds, which is approximately 3×10^{16} years. However, it is possible to write recursive methods for computing Fibonacci numbers that have O(*n*) performance. We show one such method next.

```
/** Recursive O(n) method to calculate Fibonacci numbers
    (in RecursiveMethods.java).
    pre: n >= 1
    @param fibCurrent The current Fibonacci number
    @param fibPrevious The previous Fibonacci number
    @param n The count of Fibonacci numbers left to calculate
    @return The value of the Fibonacci number calculated so far
 */
private static int fibo(int fibCurrent, int fibPrevious, int n) {
    if (n == 1)
        return fibCurrent;
    else
        return fibo(fibCurrent + fibPrevious, fibCurrent, n - 1);
}
```

Unlike method `fibonacci`, method `fibo` does not follow naturally from the recursive definition of the Fibonacci sequence. In the method `fibo`, the first argument is always the current Fibonacci number and the second argument is the previous one. We update these values for

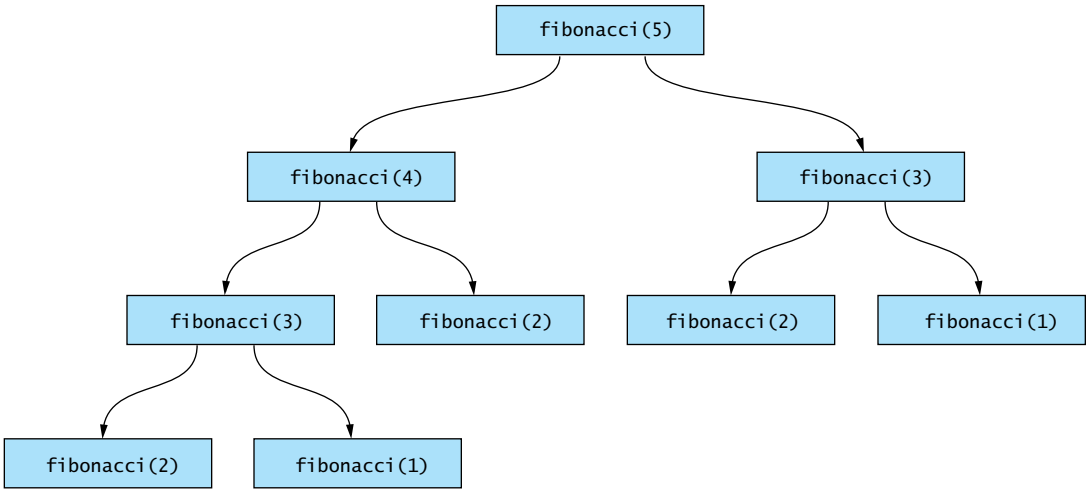


FIGURE 5.6
Method Calls Resulting from `fibonacci(5)`

each new call. When `n` is 1 (the base case), we have calculated the required Fibonacci number, so we return its value (`fibCurrent`). The recursive case

```
return fibo(fibCurrent + fibPrevious, fibCurrent, n - 1);
```

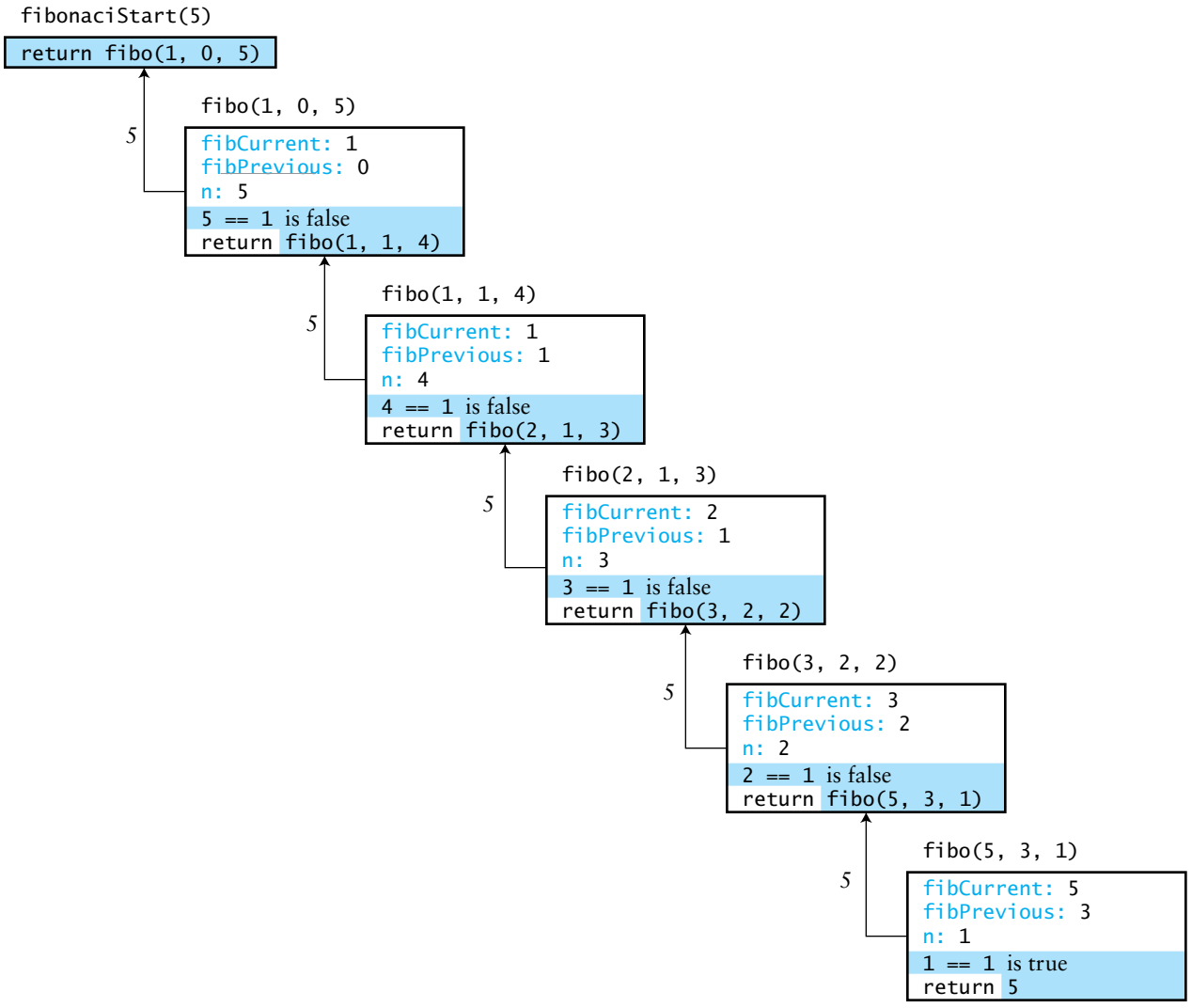
passes the sum of the current Fibonacci number and the previous Fibonacci number to the first parameter (the new value of `fibCurrent`); it passes the current Fibonacci number to the second parameter (the new value of `fibPrevious`); and it decrements `n`, making progress toward the base case.

To start this method executing, we need the following *wrapper method*, which is not recursive. This method is called a wrapper method because its main purpose is to call the recursive method and return its result. Its parameter, `n`, specifies the position in the Fibonacci sequence of the number we want to calculate. After testing for the special case `n` equals 0, it calls the recursive method `fibo`, passing the first Fibonacci number as its first argument and `n` as its third.

```
/** Wrapper method for calculating Fibonacci numbers
    (in RecursiveMethods.java).
    pre: n >= 0
    @param n The position of the desired Fibonacci number
    @return The value of the nth Fibonacci number
 */
public static int fibonacciStart(int n) {
    if (n == 0)
        return 0;
    else
        return fibo(1, 0, n);
}
```

Figure 5.7 traces the execution of the method call `fibonacciStart(5)`. Note that the first arguments for the method calls to `fibo` form the sequence 1, 1, 2, 3, 5, which is the Fibonacci sequence. Also note that the result of the first return (5) is simply passed on by each successive return. That is because the recursive case does not specify any operations other than returning the result of the next call. Note that the method `fibo` is an example of tail recursion.

FIGURE 5.7 Trace of `fibonacciStart(5)`



EXERCISES FOR SECTION 5.2

SELF-CHECK

1. Does the recursive algorithm for raising x to the power n work for negative values of n? Does it work for negative values of x? Indicate what happens if it is called for each of these cases.
2. See for what value of n the method `fibonacci` begins to take a long time to run on your computer (over 1 minute). Compare the performance of `fibonacciStart` with `fibo` for this same value.
3. Trace the execution of the following using activation frames.

```
gcd(33, 12)
gcd(12, 33)
gcd(11, 5)
```

4. For each of the following method calls, show the argument values in the activation frames that would be pushed onto the run-time stack.
   a. `gcd(6, 24)`
   b. `factorial(6)`
   c. `gcd(31, 7)`
   d. `fibonacci(6)`
   e. `fibonacciStart(7)`

PROGRAMMING

1 Write a recursive method for raising x to the power n that works for negative n as well as positive n. Use the fact that n. Use the fact that $x^{-n} = \frac{1}{x^n}$.

2 Modify the factorial method to throw an `IllegalArgumentException` if n is negative.

3 Modify the `Fibonacci` method to throw an `IllegalArgumentException` if its argument is less than or equal to zero.

4 Write a class that has an iterative method for calculating Fibonacci numbers. Use an array that saves each Fibonacci number as it is calculated. Your method should take advantage of the existence of this array so that subsequent calls to the method simply retrieve the desired Fibonacci number if it has been calculated. If not, start with the largest Fibonacci number in the array rather than repeating all calculations.

5.3 Recursive Array Search

Searching an array is an activity that can be accomplished using recursion. The simplest way to search an array is a *linear search*. In a linear search, we examine one array element at a time, starting with the first element or the last element, to see whether it matches the target. The array element we are seeking may be anywhere in the array, so on average we will examine $\frac{n}{2}$ items to find the target if it is in the array. If it is not in the array, we will have to examine all n elements (the worst case). This means linear search is an $O(n)$ algorithm.

Design of a Recursive Linear Search Algorithm

Let's consider how we might write a recursive algorithm for an array search that returns the subscript of a target item.

The base case would be an empty array. If the array is empty, the target cannot be there, so the result should be –1. If the array is not empty, we will assume that we can examine just the first element of the array, so another base case would be when the first array element matches the target. If so, the result should be the subscript of the first array element.

The recursive step would be to search the rest of the array, excluding the first element. So our recursive step should search for the target starting with the current second array element, which will become the first element in the next execution of the recursive step. The algorithm follows.

Algorithm for Recursive Linear Array Search

1. `if` the array is empty
2. The result is –1.

`else if` the first element matches the target

3. The result is the subscript of the first element.

`else`

4. Search the array excluding the first element and return the result.

Implementation of Linear Search

The following method, `linearSearch (part of class RecursiveMethods)`, shows the linear search algorithm.

```
/** Recursive linear search method (in RecursiveMethods.java).
    @param items The array being searched
    @param target The item being searched for
    @param posFirst The position of the current first element
    @return The subscript of target if found; otherwise -1
 */
private static int linearSearch(Object[] items,
                                Object target, int posFirst) {
    if (posFirst == items.length)
        return -1;
    else if (target.equals(items[posFirst]))
        return posFirst;
    else
        return linearSearch(items, target, posFirst + 1);
}
```

The method parameter `posFirst` represents the subscript of the current first element. The first condition tests whether the array left to search is empty. The condition `(posFirst == items.length)` is **true** when the subscript of the current first element is beyond the bounds of the array. If so, method `linearSearch` returns `-1`. The statement

```
return linearSearch(items, target, posFirst + 1);
```

implements the recursive step; it increments `posFirst` to exclude the current first element from the next search.

To search an array x for target, you could use the method call

```
RecursiveMethods.linearSearch(x, target, 0)
```

However, since the third argument would always be 0, we can define a non-recursive wrapper method (also called `linearSearch`) that has just two parameters: items and target.

```
/** Wrapper for recursive linear search method (in
    RecursiveMethods.java).
    @param items The array being searched
    @param target The object being searched for
    @return The subscript of target if found; otherwise -1
 */
public static int linearSearch(Object[] items, Object target) {
    return linearSearch(items, target, 0);
}
```

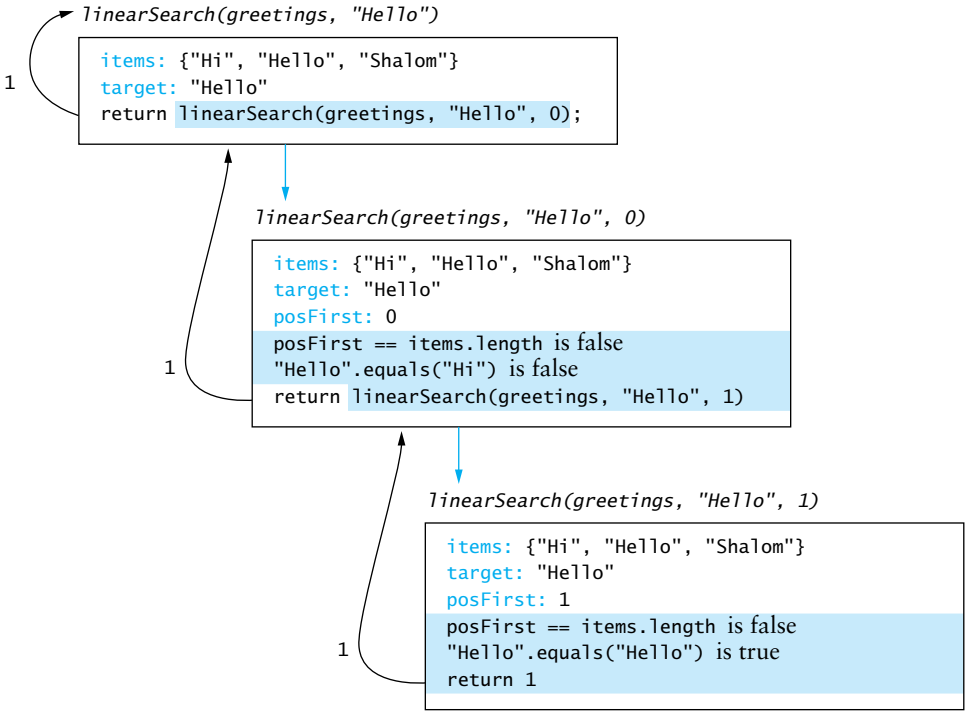
The sole purpose of this method is to call the recursive method, passing on its arguments with 0 as a third argument, and return its result. This method definition overloads the previous one, which has private visibility.

Figure 5.8 traces the execution of the call to linearSearch in the second statement.

```
String[] greetings = {"Hi", "Hello", "Shalom"};
int posHello = linearSearch(greetings, "Hello");
```

The value returned to `posHello` will be 1.

FIGURE 5.8 Trace of `linearSearch(greetings, "Hello")`



Design of a Binary Search Algorithm

A second approach to searching an array is called *binary search*. Binary search can be performed only on an array that has been sorted. In binary search, the stopping cases are the same as for linear search:

- When the array is empty.
- When the array element being examined matches the target.

However, rather than examining the last array element, binary search compares the "middle" element of the array to the target. If there is a match, it returns the position of the middle element. Otherwise, because the array has been sorted, we know with certainty which half of the array must be searched to find the target. We then can exclude the other half of the array (not just one element as with linear search). The binary search algorithm (first introduced in Section 5.1) follows.

Binary Search Algorithm

1. **if** the array is empty
2. Return –1 as the search result.

   else if the middle element matches the target

3. Return the subscript of the middle element as the result.

else if the target is less than the middle element

4. Recursively search the array elements before the middle element and return the result.

else

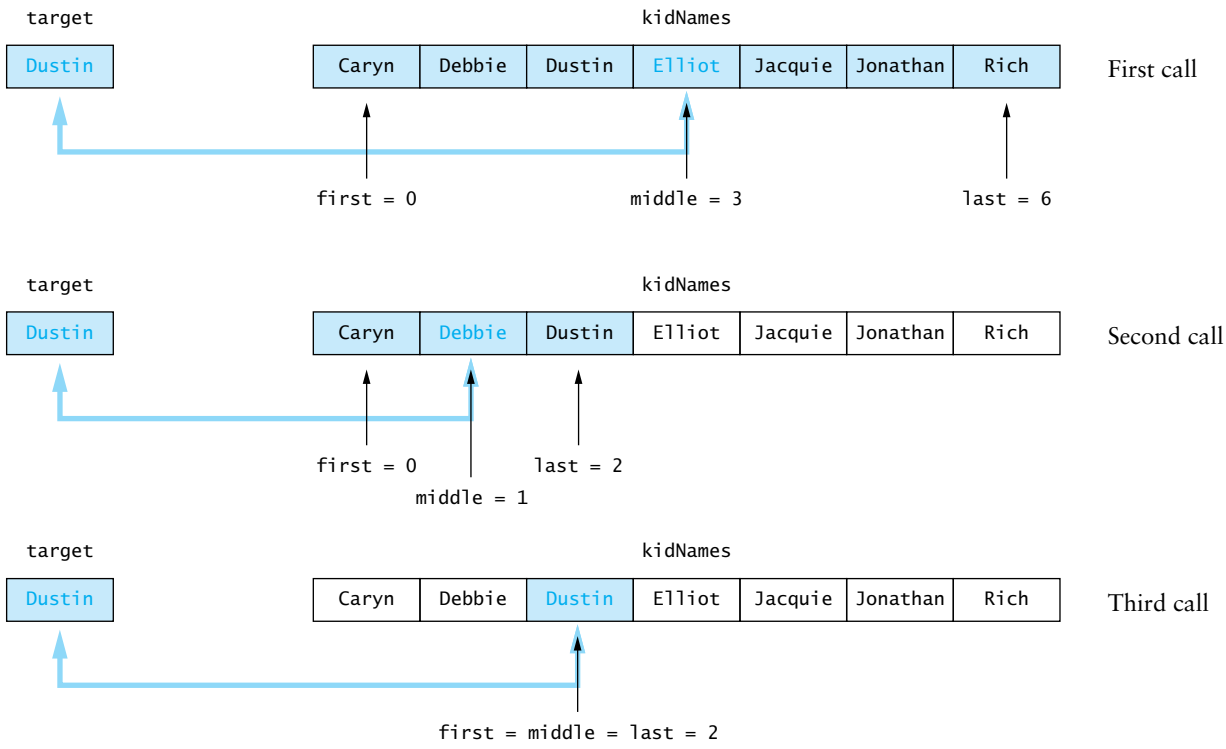
5. Recursively search the array elements after the middle element and return the result.

Figure 5.9 illustrates binary search for an array with seven elements. The shaded array elements are the ones that are being searched each time. The array element in blue is the one that is being compared to the target. In the first call, we compare `"Dustin"` to `"Elliot"`. Because `"Dustin"` is smaller, we need to search only the part of the array before `"Elliot"` (consisting of just three candidates). In the second call, we compare `"Dustin"` to `"Debbie"`. Because `"Dustin"` is larger, we need to search only the shaded part of the array after `"Debbie"` (consisting of just one candidate). In the third call, we compare `"Dustin"` to `"Dustin"`, and the subscript of `"Dustin"` (2) is our result. If there were no match at this point (e.g., the array contained `"Duncan"` instead of `"Dustin"`), the array of candidates to search would become an empty array.

Efficiency of Binary Search

Because we eliminate at least half of the array elements from consideration with each recursive call, binary search is an O(log *n*) algorithm. To verify this, an unsuccessful search of an array of size 16 could result in our searching arrays of size 16, 8, 4, 2, and 1 to determine that

FIGURE 5.9
Binary Search for `"Dustin"`



the target was not present. Thus, an array of size 16 requires a total of 5 probes in the worst case (16 is 2^4, so 5 is $\log_2 16 + 1$). If we double the array size, we would need to make only 6 probes for an array of size 32 in the worst case (32 is 2^5, so 6 is $\log_2 32 + 1$). The advantages of binary search become even more apparent for larger arrays. For an array with 32,768 elements, the maximum number of probes required would be 16 ($\log_2 32{,}768$ is 15), and if we expand the array to 65,536 elements, we would increase the number of probes required only to 17.

The Comparable Interface

We introduced the `Comparable` interface in Section 2.8. Classes that implement this interface must define a `compareTo` method that enables its objects to be compared in a standard way. The method `compareTo` returns an integer whose value indicates the relative ordering of the two objects being compared (as described in the `@return` tag below). If the target is type `Comparable`, we can apply its `compareTo` method to compare the target to the objects stored in the array. `T` represents the type of the object being compared.

```
/** Instances of classes that realize this interface can be
    compared.
    @param <T> The type of object this object can be compared to.
 */
public interface Comparable<T> {
    /** Method to compare this object to the argument object.
        @param obj The argument object
        @return Returns a negative integer if this object < obj;
                zero if this object equals obj;
                a positive integer if this object > obj
     */
    int compareTo(T obj);
}
```

Implementation of Binary Search

Listing 5.1 shows a recursive implementation of the binary search algorithm and its non-recursive wrapper method. The parameters `first` and `last` are the subscripts of the first element and last element in the array being searched. For the initial call to the recursive method from the wrapper method, `first` is 0 and last is `items.length - 1`. The parameter `target` is type `Comparable`.

The condition (`first > last`) becomes true when the list of candidates is empty. The statement

```
int middle = (first + last) / 2;
```

computes the subscript of the "middle" element in the current array (midway between `first` and `last`).

The statement

```
int compResult = target.compareTo(items[middle]);
```

saves the result of comparing the target to the middle element of the array. If the result is 0 (a match), the subscript middle is returned. If the result is negative, the recursive step

```
return binarySearch(items, target, first, middle - 1);
```

returns the result of searching the part of the current array before the middle item (with subscripts `first` through `middle - 1`). If the result is positive, the recursive step

```
return binarySearch(items, target, middle + 1, last);
```

returns the result of searching the part of the current array after the middle item (with subscripts `middle + 1` through `last`).

LISTING 5.1

Method `binarySearch`

```
/** Recursive binary search method (in RecursiveMethods.java).
    @param <T> The item type
    @param items The array being searched
    @param target The object being searched for
    @param first The subscript of the first element
    @param last The subscript of the last element
    @return The subscript of target if found; otherwise -1.
 */
private static <T extends Comparable<T>> int binarySearch(T[] items, T target,
                                    int first, int last) {
    if (first > last)
        return -1;     // Base case for unsuccessful search.
    else {
        int middle = (first + last) / 2; // Next probe index.
        int compResult = target.compareTo(items[middle]);
        if (compResult == 0)
            return middle; // Base case for successful search.
        else if (compResult < 0)
            return binarySearch(items, target, first, middle - 1);
        else
            return binarySearch(items, target, middle + 1, last);
    }
}

/** Wrapper for recursive binary search method (in RecursiveMethods.java).
    @param <T> The item type.
    @param items The array being searched
    @param target The object being searched for
    @return The subscript of target if found; otherwise -1.
 */
public static <T extends Comparable<T>> int binarySearch(T[] items, T target) {
    return binarySearch(items, target, 0, items.length - 1);
}
```

Figure 5.10 traces the execution of `binarySearch` for the array shown in Figure 5.9. The parameter `items` always references the same array; however, the pool of candidates changes with each call.

SYNTAX Declaring a Generic Method

FORM:

methodModifiers <genericParameters> returnType methodName(methodParameters)

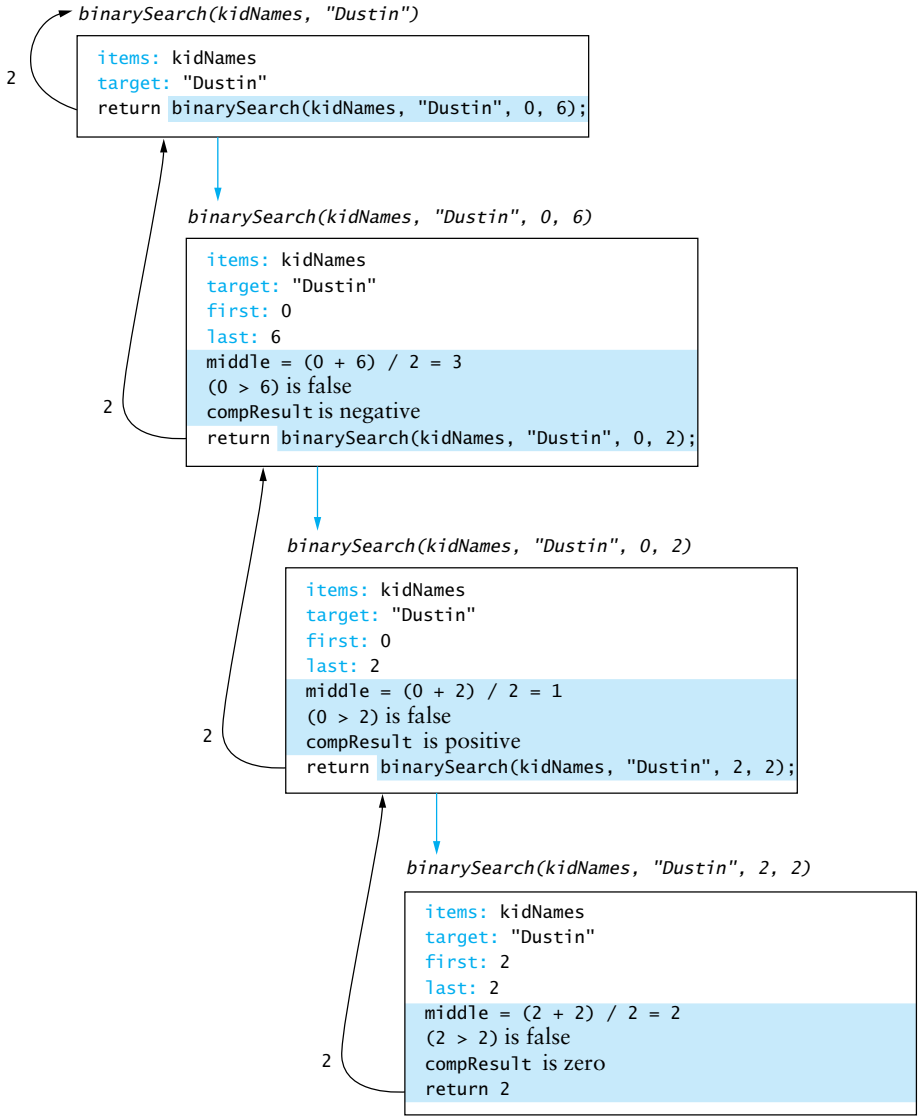
EXAMPLE

```
public static <T extends Comparable<T>> int binarySearch(T[] items, T target)
```

MEANING

To declare a generic method, list the *genericParameters* inside the symbol pair <> and between the *methodModifers* (e.g., public static) and the *returnType*. The *genericParameters* can then be used in the specification of the *methodParameters* and in the method body.

FIGURE 5.10
Trace of `binarySearch(kidNames, "Dustin")`



Testing Binary Search

To test the binary search algorithm, you must test arrays with an even number of elements and arrays with an odd number of elements. You must also test arrays that have duplicate items. Each array must be tested for the following cases:

- The target is the element at each position of the array, starting with the first position and ending with the last position.
- The target is less than the smallest array element.
- The target is greater than the largest array element.
- The target is a value between each pair of items in the array.

Method Arrays.binarySearch

The Java API class `Arrays` contains a `binarySearch` method. It can be called with sorted arrays of primitive types or with sorted arrays of objects. If the objects in the array are not mutually comparable or if the array is not sorted, the results are undefined. If there are multiple copies of the target value in the array, there is no guarantee as to which one will be found. This is the same as for our `binarySearch` method. The method throws a `ClassCastException` if the target is not comparable to the array elements (e.g., if the target is type `Integer` and the array elements are type `String`).

EXERCISES FOR SECTION 5.3

SELF-CHECK

1. For the array shown in Figure 5.9, show the values of `first`, `last`, `middle`, and `compResult` in successive frames when searching for a target of `"Rich"`; when searching for a target of `"Alice"`; and when searching for a target of `"Daryn"`.
2. How many elements will be compared to target for an unsuccessful binary search in an array of 1000 items? What is the answer for 2000 items?
3. If there are multiple occurrences of the target item in an array, what can you say about the subscript value that will be returned by `linearSearch`? Answer the same question for `binarySearch`.
4. Write a recursive algorithm to find the largest value in an array of integers.
5. Write a recursive algorithm that searches a string for a target character and returns the position of its first occurrence if it is present or –1 if it is not.

PROGRAMMING

1. Write a recursive method to find the sum of all values stored in an array of integers.
2. Write a recursive linear search method with a recursive step that finds the last occurrence of a target in an array, not the first. You will need to modify the linear search method so that the last element of the array is always tested, not the first. You will need to pass the current length of the array as an argument.
3. Implement the method for Self-Check Exercise 4.
4. Implement the method for Self-Check Exercise 5. You will need to keep track of the current position in the string through a method parameter.

5.4 Recursive Data Structures

Computer scientists often encounter data structures that are defined recursively. A *recursive data structure* is one that has another version of itself as a component. We will define the tree data structure as a recursive data structure in Chapter 6, but we can also define a linked list, described in Chapter 2, as a recursive data structure. In this section, we demonstrate that recursive methods provide a very natural mechanism for processing recursive data structures. The first language developed for artificial intelligence research was a recursive language designed expressly for LISt Processing and therefore called LISP.

Recursive Definition of a Linked List

The following definition implies that a nonempty linked list is a collection of nodes such that each node references another linked list consisting of the nodes that follow it in the list. The last node references an empty list.

A linked list is empty, or it consists of a node, called the list head, that stores data and a reference to a linked list.

Class `LinkedListRec`

We will define a class `LinkedListRec<E>` that implements several list operations using recursive methods. The class `LinkedListRec<E>` has a private inner class called `Node<E>`, which is defined in Listing 2.1. A `Node<E>` object has attributes `data (type E)` and `next (type Node)`. `Class LinkedListRec<E>` has a single data field head (`data type Node<E>`).

```
/** A recursive linked list class with recursive methods. */
public class LinkedListRec<E> {

    /** The list head */
    private Node<E> head;

    // Insert inner class Node<E> here. See Listing 2.1.
    . . .
}
```

We will write the following recursive methods: `size` (returns the size), `toString` (represents the list contents as a string), `add` (adds an element to the end of the list), and replace (replaces one object in a list with another). We code each operation using a pair of methods: a public wrapper method that calls a private recursive method. To perform a list operation, you apply a wrapper method to an instance of class `LinkedListRec`.

Method `size`

The method `size` returns the size of a linked list and is similar to the method `length` defined earlier for a string. The recursive method returns 0 if the list is empty (`head == null` is true). Otherwise, the statement

```
return 1 + size(head.next);
```

returns 1 plus the size of the rest of the list that is referenced by `head.next`.

The wrapper method calls the recursive method, passing the list head as an argument, and returns the value returned by the recursive method. In the initial call to the recursive method, head will reference the first list node. In each subsequent call, `head` will reference the successor of the node that it currently references.

```
/** Finds the size of a list.
    @param head The head of the current list
    @return The size of the current list
 */
private int size(Node<E> head) {
    if (head == null)
        return 0;
    else
        return 1 + size(head.next);
}

/** Wrapper method for finding the size of a list.
    @return The size of the list
 */
public int size() {
    return size(head);
}
```

Method toString

The method `toString` returns a string representation of a linked list. The recursive method is very similar to the method size. The statement

```
return head.data + "\n" + toString(head.next);
```

appends the data in the current list head to the string representation of the rest of the list. The line space character is inserted after each list item. If the list contains the elements `"hat"`, `"55"`, and `"dog"`, the string result would be `"hat\n55\ndog\n"`.

```
/** Returns the string representation of a list.
    @param head The head of the current list
    @return The state of the current list
 */
private String toString(Node<E> head) {
    if (head == null)
return "";
    else
        return head.data + "\n" + toString(head.next);
}
/** Wrapper method for returning the string representation of a list.
    @return The string representation of the list
 */
public String toString() {
    return toString(head);
}
```

Method replace

The method `replace` replaces each occurrence of an object in a list (parameter `oldobj`) with a different object (parameter `newobj`). The **`if`** statement in the recursive method is different from what we are used to. The method does nothing for the base case of an empty list. If the list is not empty, the **`if`** statement

```
if (oldobj.equals(head.data))
head.data = newobj;
```

tests whether the item in the current list head matches `oldobj`. If so, it stores `newobj` in the current list head. Regardless of whether or not a replacement is performed, the method `replace` is called recursively to process the rest of the list.

```
/** Replaces all occurrences of oldobj with newobj.
    post: Each occurrence of oldobj has been replaced by newobj.
    @param head The head of the current list
    @param oldobj The object being removed
    @param newobj The object being inserted
 */
private void replace(Node<E> head, E oldObj, E newObj) {
    if (head != null) {
        if (oldobj.equals(head.data))
            head.data = newobj;
        replace(head.next, oldobj, newobj);
    }
}

/** Wrapper method for replacing oldobj with newobj.
    post: Each occurrence of oldobj has been replaced by newobj.
    @param oldobj The object being removed
    @param newobj The object being inserted
 */
public void replace(E oldobj, E newobj) {
    replace(head, oldobj, newobj);
}
```

Method add

You can use the add method to add nodes to an existing list. You can also use it to build a list by adding new nodes to the end of an initially empty list.

The add methods have two features that are different from what we have seen before. The wrapper method tests for an empty list (head == null is **true**), and it calls the recursive add method only if the list is not empty. If the list is empty, the wrapper add method creates a new node, which is referenced by the data field head, and stores the first list item in this node.

```
/** Adds a new node to the end of a list.
    @param head The head of the current list
    @param data The data for the new node
 */
private void add(Node<E> head, E data) {
    // If the list has just one element, add to it.
    if (head.next == null)
        head.next = new Node<>(data);
    else
        add(head.next, data); // Add to rest of list.
}

/** Wrapper method for adding a new node to the end of a list.
    @param data The data for the new node
 */
public void add(E data) {
    if (head == null)
        head = new Node<>(data); // List has 1 node.
    else
        add(head, data);
}
```

For each node referenced by argument head, the recursive method tests to see whether the node referenced by argument head is the last node in the list (head.next is null). If so, the method add then resets head.next to reference a new node that contains the data being inserted.

PITFALL

Testing for an Empty List Instead of Testing for the Last List Node

In the recursive method add, we test whether head.next is null. This condition is true when head references a list with just one node. We then reset its next field to reference a new node. If we tested whether head was null (an empty list) and then executed the statement

```
head = new Node<>(data);
```

this would have no effect on the original list. The local reference head would be changed to reference the new node, but this node would not be connected to a node in the original list.

Removing a List Node

One of the reasons for using linked lists is that they enable easy insertion and removal of nodes. We show how to do removal next and leave insertion as an exercise. In the following recursive method remove, the first base case returns **false** if the list is empty. The second base case determines whether the list head should be removed by comparing its data field to outData. If there is a match, the assignment statement removes the list head by connecting

its predecessor (referenced by pred) to the successor of the list head. For this case, method remove returns **true**. The recursive case applies remove to the rest of the list. In the next execution of the recursive method, the current list head will be referenced by pred, and the successor of the current list head will be referenced by head.

```
/** Removes a node from a list.
    post: The first occurrence of outData is removed.
    @param head The head of the current list
    @param pred The predecessor of the list head
    @param outData The data to be removed
    @return true if the item is removed
            and false otherwise
 */
private boolean remove(Node<E> head, Node<E> pred, E outData) {
    if (head == null) // Base case - empty list
        return false;
    else if (head.data.equals(outData)) { // 2nd base case
        pred.next = head.next; // Remove head.
        return true;
    } else
        return remove(head.next, head, outData);
}
```

The following wrapper method takes care of the special case where the node to be removed is at the head of the list. The first condition returns **false** if the list is empty. The second condition removes the list head and returns **true** if the list head contains the data to be removed. The **else** clause calls the recursive remove method. In the first execution of the recursive method, head will reference the actual second node and pred will reference the actual first node.

```
/** Wrapper method for removing a node (in LinkedListRec).
    post: The first occurrence of outData is removed.
    @param outData The data to be removed
    @return true if the item is removed,
            and false otherwise
 */
public boolean remove(E outData) {
    if (head == null)
        return false;
    else if (head.data.equals(outData)) {
        head = head.next;
        return true;
    } else
        return remove(head.next, head, outData);
}
```

EXERCISES FOR SECTION 5.4

SELF-CHECK

1. Describe the result of executing each of the following statements:

```
LinkedListRec<String> aList = new LinkedListRec<String>();
aList.add("bye");
aList.add("hello");
System.out.println(aList.size() + ", " + aList.toString());
aList.replace("hello", "welcome");
aList.add("OK");
aList.remove("bye");
aList.remove("hello");
System.out.println(aList.size() + ", " + aList.toString());
```

2. Trace each call to a `LinkedListRec` method in Exercise 1 above.
3. Write a recursive algorithm for method `insert(E obj, int index)` where `index` is the position of the insertion.
4. Write a recursive algorithm for method `remove(int index)` where `index` is the position of the item to be removed.

PROGRAMMING

1. Write an equals method for the `LinkedListRec` class that compares this `LinkedListRec` object to one specified by its argument. Two lists are equal if they have the same number of nodes and store the same information at each node. Don't use the `size` method.
2. Write a search method that returns **`true`** if its argument is stored as the data field of a `LinkedListRec` node and returns **`false`** if its argument is not stored in any node.
3. Write a recursive method `insertBefore` that inserts a specified data object before the first occurrence of another specified data object. For example, the method call `aList.insertBefore(target, inData)` would insert the object referenced by `inData` in a new node just before the first node of `aList` that stores a reference to `target` as its data.
4. Write a recursive method `reverse` that reverses the elements in a linked list.
5. Code method `insert` in Self-Check Exercise 3.
6. Code method `remove` in Self-Check Exercise 4.

5.5 Problem Solving with Recursion

In this section, we discuss recursive solutions to two problems. Our recursive solutions will break each problem up into multiple smaller versions of the original problem. Both problems are easier to solve using recursion because recursive thinking enables us to split each problem into more manageable subproblems. They would both be much more difficult to solve without recursion.

CASE STUDY Towers of Hanoi

Problem You may be familiar with a version of this problem that is sold as a child's puzzle. There is a board with three pegs and three disks of different sizes (see Figure 5.11). The goal of the game is to move the three disks from the peg where they have been placed (largest disk on the bottom, smallest disk on the top) to one of the empty pegs, subject to the following constraints:

- Only the top disk on a peg can be moved to another peg.
- A larger disk cannot be placed on top of a smaller disk.

Analysis We can solve this problem by displaying a list of moves to be made. The problem inputs will be the number of disks to move, the starting peg, the destination peg, and the temporary peg. Table 5.1 shows the problem inputs and outputs. We will write a class `Tower` that contains a method `showMoves` that builds a string with all the moves.

FIGURE 5.11
Children's Version of Towers of Hanoi

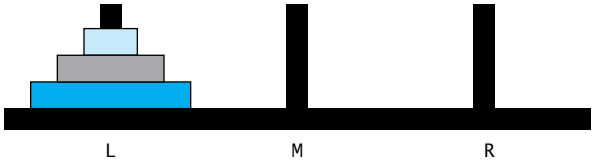


TABLE 5.1
Inputs and Outputs for Towers of Hanoi Problem

Problem Inputs
Number of disks (an integer)
Letter of starting peg: L (left), M (middle), or R (right)
Letter of destination peg: (L, M, or R), but different from starting peg
Letter of temporary peg: (L, M, or R), but different from starting peg and destination peg
Problem Outputs
A list of moves

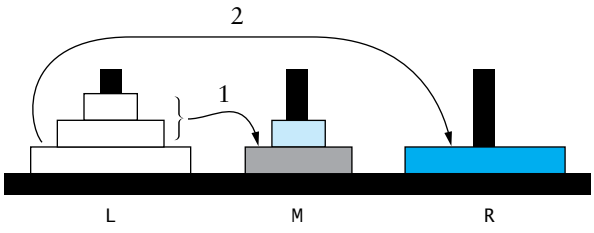
Design

We still need to determine a strategy for making a move. If we examine the situation in Figure 5.11 (all three disks on the L peg), we can derive a strategy to solve it. If we can figure out how to move the top two disks to the M peg (a two-disk version of the original problem), we can then place the bottom disk on the R peg (see Figure 5.12). Now all we need to do is move the two disks on the M peg to the R peg. If we can solve both of these two-disk problems, then the three-disk problem is also solved.

Solution to Two-Disk Problem: Move Three Disks from Peg L to Peg R

1. Move the top two disks from peg L to peg M.
2. Move the bottom disk from peg L to peg R.
3. Move the top two disks from peg M to peg R.

FIGURE 5.12
Towers of Hanoi after the First Two Steps in Solution of the Three-Disk Problem



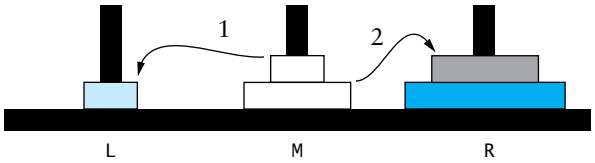
We can split the solution to each two-disk problem into three problems involving single disks. We solve the second two-disk problem next; the solution to the first one (move the top two disks from peg L to peg M) is quite similar.

Solution to Two-Disk Problem: Move Top Two Disks from Peg M to Peg R

1. Move the top disk from peg M to peg L.
2. Move the bottom disk from peg M to peg R.
3. Move the top disk from peg L to peg R.

In Figure 5.13, we show the pegs after Steps 1 and 2. When Step 3 is completed, the three pegs will be on peg R.

FIGURE 5.13
Towers of Hanoi after First Two Steps in Solution of the Two-Disk Problem

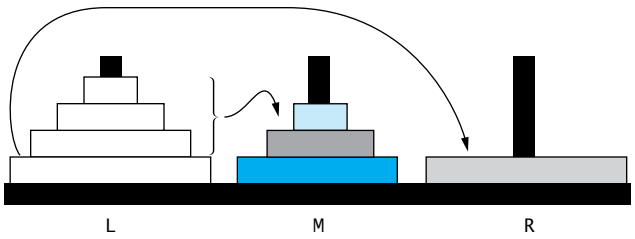


In a similar way, we can split a four-disk problem into two three-disk problems. Figure 5.14 shows the pegs after the top three disks have been moved from peg L to peg M. Because we know how to solve three-disk problems, we can also solve four-disk problems.

Solution to Four-Disk Problem: Move Four Disks from Peg L to Peg R

1. Move the top three disks from peg L to peg M.
2. Move the bottom disk from peg L to peg R.
3. Move the top three disks from peg M to peg R.

FIGURE 5.14
Towers of Hanoi after the First Two Steps in Solution of the Four-Disk Problem



Next, we show a general recursive algorithm for moving *n* disks from one of the three pegs to a different peg.

Recursive Algorithm for *n*-Disk Problem: Move *n* Disks from the Starting Peg to the Destination Peg

1. `if` *n* is 1
2. Move disk 1 (the smallest disk) from the starting peg to the destination peg.
3. `else`
4. Move the top *n* – 1 disks from the starting peg to the temporary peg (neither starting nor destination peg).
5. Move disk *n* (the disk at the bottom) from the starting peg to the destination peg.
6. Move the top *n* – 1 disks from the temporary peg to the destination peg.

The stopping case is the one-disk problem. The recursive step enables us to split the *n*-disk problem into two (*n* – 1) disk problems and a single-disk problem. Each problem has a different starting peg and destination peg.

Our recursive solution method `showMoves` will display the solution as a list of disk moves. For each move, we show the number of the disk being moved and its starting and destination pegs. For example, for the two-disk problem shown earlier (move two disks from the middle peg, M, to the right peg, R), the list of moves would be

```
Move disk 1 from peg M to peg L
Move disk 2 from peg M to peg R
Move disk 1 from peg L to peg R
```

The method `showMoves` must have the number of disks, the starting peg, the destination peg, and the temporary peg as its parameters. If there are *n* disks, the bottom disk has number *n* (the top disk has number 1). Table 5.2 describes the method required for class `TowersofHanoi`.

Implementation

Listing 5.2 shows class `TowersofHanoi`. In method `showMoves`, the recursive step

```
return showMoves(n - 1, startPeg, tempPeg, destPeg)
       + "Move disk " + n + " from peg " + startPeg
       + " to peg " + destPeg + "\n"
       + showMoves(n - 1, tempPeg, destPeg, startPeg);
```

TABLE 5.2
Class `TowersofHanoi`

Method	Behavior
`public String showMoves(int n, char startPeg, char destPeg, char tempPeg)`	Builds a string containing all moves for a game with `n` disks on `startPeg` that will be moved to `destPeg` using `tempPeg` for temporary storage of disks being moved

returns the string formed by concatenating the list of moves for the first (`n` – 1)-disk problem (the recursive call after return), the move required for the bottom disk (disk `n`), and the list of moves for the second (`n` – 1)-disk problem.

LISTING 5.2
Class TowersofHanoi

```
/** Class that solves Towers of Hanoi problem. */
public class TowersofHanoi {
    /** Recursive method for "moving" disks.
        pre: startPeg, destPeg, tempPeg are different.
        @param n is the number of disks
        @param startPeg is the starting peg
        @param destPeg is the destination peg
        @param tempPeg is the temporary peg
        @return A string with all the required disk moves
     */
    public static String showMoves(int n, char startPeg,
                                   char destPeg, char tempPeg) {
        if (n == 1) {
            return "Move disk 1 from peg " + startPeg +
                   " to peg " + destPeg + "\n";
        } else { // Recursive step
            return showMoves(n - 1, startPeg, tempPeg, destPeg)
                   + "Move disk " + n + " from peg " + startPeg
                   + " to peg " + destPeg + "\n"
                   + showMoves(n - 1, tempPeg, destPeg, startPeg);
        }
    }
}
```

Testing

Figure 5.15 shows the result of executing the following `main` method for the data 3, L, and R ("move 3 disks from peg L to peg R"). The first three lines are the solution to the problem "move 2 disks from peg L to peg M," and the last three lines are the solution to the problem "move 2 disks from peg M to peg R."

```
public static void main(String[] args) {
     Scanner in = new Scanner(System.in);
     System.out.print("Enter number of disks ");
     int nDisks = in.nextInt();
     String moves = showMoves(nDisks, 'L', 'R', 'M');
     System.out.println(moves);
}
```

FIGURE 5.15
Solution to "Move 3 Disks from Peg L to Peg R"

```
Enter number of disks 3
Move disk 1 from peg L to peg R
Move disk 2 from peg L to peg M
Move disk 1 from peg R to peg M
Move disk 3 from peg L to peg R
Move disk 1 from peg M to peg L
Move disk 2 from peg M to peg R
Move disk 1 from peg L to peg R
```

Visualization of Towers of Hanoi

We have provided a graphical visualization that you can use to observe the movement of disks in a solution to the Towers of Hanoi. You can access it through the textbook Web site for this book.

CASE STUDY Counting Cells in a Blob

In this case study, we consider how we might process an image that is presented as a two-dimensional array of color values. The information in the two-dimensional array might come from a variety of sources. For example, it could be an image of part of a person's body that comes from an X-ray or an MRI, or it could be a picture of part of the earth's surface taken by a satellite. Our goal in this case study is to determine the size of any area in the image that is considered abnormal because of its color values.

Problem

You have a two-dimensional grid of cells, and each cell contains either a normal background color or a second color, which indicates the presence of an abnormality. The user wants to know the size of a *blob*, where a blob is a collection of contiguous abnormal cells. The user will enter the *x*, *y* coordinates of a cell in the blob, and the count of all cells in that blob will be determined.

Analysis Data Requirements

PROBLEM INPUTS

- The two-dimensional grid of cells
- The coordinates of a cell in a blob

PROBLEM OUTPUTS

- The count of cells in the blob

Classes

We will have two classes. Class `TwoDimGrid` will manage the two-dimensional grid of cells. You can find the discussion of the design and implementation of this class on the Web site for this book. Here we will focus on the design of class `Blob`, which contains the recursive method that counts the number of cells in a blob.

Design

Table 5.3 describes the public methods of class `TwoDimGrid`, and Table 5.4 describes class `Blob`.

Method `countCells` in class `Blob` is a recursive method that is applied to a `TwoDimGrid` object. Its parameters are the `(x, y)` position of a cell. The algorithm follows.

TABLE 5.3
Class `TwoDimGrid`

Method	Behavior
`void recolor(int x, int y, Color aColor)`	Resets the color of the cell at position (x, y) to aColor
`Color getColor(int x, int y)`	Retrieves the color of the cell at position (x, y)
`int getNRows()`	Returns the number of cells in the *y*-axis
`int getNCols()`	Returns the number of cells in the *x*-axis

TABLE 5.4
Class `Blob`

Method	Behavior
`int countCells(int x, int y)`	Returns the number of cells in the blob at (x, y)

Algorithm for countCells(x, y)

1. `if` the cell at (x, y) is outside the grid
2. The result is 0.

`else if` the color of the cell at (x, y) is not the abnormal color

3. The result is 0.

`else`

4. Set the color of the cell at (x, y) to a temporary color.
5. The result is 1 plus the number of cells in each piece of the blob that includes a nearest neighbor.

The two stopping cases are reached if the coordinates of the cell are out of bounds or if the cell does not have the abnormal color and, therefore, can't be part of a blob. The recursive step involves counting 1 for a cell that has the abnormal color and adding the counts for the blobs that include each immediate neighbor cell. Each cell has eight immediate neighbors: two in the horizontal direction, two in the vertical direction, and four in the diagonal directions.

If no neighbor has the abnormal color, then the result will be just 1. If any neighbor cell has the abnormal color, then it will be counted along with all its neighbor cells that have the abnormal color, and so on until no neighboring cells with abnormal color are encountered (or the edge of the grid is reached). The reason for setting the color of the cell at (x, y) to a temporary color is to prevent it from being counted again when its neighbors' blobs are counted.

Implementation

Listing 5.3 shows class Blob. The interface `GridColors` defines the three constants: `BACKGROUND`, `ABNORMAL`, and `IN_BLOB`. We make these constants available by using the static import statement:

```
import static GridColors.*;
```

The first terminating condition,

```
(x < 0 || x >= grid.getNCols() || y < 0 || y >= grid.getNRows())
```

compares `x` to 0 and the value returned by `getNCols()`, the number of columns in the grid. Because `x` is plotted along the horizontal axis, it is compared to the number of columns, not the number of rows. The same test is applied to `y` and the number of rows. The second terminating condition,

```
(!grid.getColor(x, y).equals(ABNORMAL))
```

is true if the cell at (x, y) has either the background color or the in-blob color.

The recursive step is implemented by the statement

```
return 1
    + countCells(x - 1, y + 1) + countCells(x, y + 1)
    + countCells(x + 1, y + 1) + countCells(x - 1, y)
    + countCells(x + 1, y) + countCells(x - 1, y - 1)
    + countCells(x, y - 1) + countCells(x + 1, y - 1);
```

Each recursive call to `countCells` has as its arguments the coordinates of a neighbor of the cell at (x, y). The value returned by each call will be the number of cells in the blob it belongs to, excluding the cell at (x, y) and any other cells that may have been counted already.

LISTING 5.3
Class Blob

```
import java.awt.*;
import static GridColors.*;

/** Class that solves problem of counting abnormal cells. */
public class Blob {

    /** The grid */
    private TwoDimGrid grid;

    /** Constructors */
    public Blob(TwoDimGrid grid) {
        this.grid = grid;
    }

    /** Finds the number of cells in the blob at (x, y).
        pre: Abnormal cells are in ABNORMAL color;
             Other cells are in BACKGROUND color.
        post: All cells in the blob are in the TEMPORARY color.
        @param x The x-coordinate of a blob cell
        @param y The y-coordinate of a blob cell
        @return The number of cells in the blob that contains (x, y)
     */
    public int countCells(int x, int y) {
        int result;

        if (x < 0 || x >= grid.getNCols()
                || y < 0 || y >= grid.getNRows())
            return 0;
        else if (!grid.getColor(x, y).equals(ABNORMAL))
            return 0;
        else {
            grid.recolor(x, y, IN_BLOB);
            return 1
                + countCells(x - 1, y + 1) + countCells(x, y + 1)
                + countCells(x + 1, y + 1) + countCells(x - 1, y)
                + countCells(x + 1, y) + countCells(x - 1, y - 1)
                + countCells(x, y - 1) + countCells(x + 1, y - 1);
        }
    }
}
```

SYNTAX import static

FORM:

```
import static Class.*;
```

or

```
import static Class.StaticMember;
```

EXAMPLE:

```
import static GridColors.*;
```

MEANING:

The static members of the class or interface *Class* are visible in the file containing the import. If * is specified, then all static members are imported, otherwise if a specific member is listed, then this member is visible.

Testing

To test the recursive algorithm in this case study and the one in the next section, we will need to implement class `TwoDimGrid`. To make the program interactive and easy to use, we implemented `TwoDimGrid` as a two-dimensional grid of buttons placed in a panel. When the button panel is placed in a frame and displayed, the user can toggle the color of a button (from normal to abnormal and back to normal) by clicking it. Similarly, the program can change the color of a button by applying the recolor method to the button. Information about the design of class `TwoDimGrid` is on the textbook Web site for this book, as is the class itself.

We also provide a class `BlobTest` on the textbook Web site. This class allows the user to load the colors for the button panel from a file that contains a representation of the image as lines of 0s and 1s, where 0 is the background color and 1 is the abnormal color. Alternatively, the user can set the dimensions of the grid and then enter the abnormal cells by clicking on each button that represents an abnormal cell. When the grid has been finalized, the user clicks twice on one of the abnormal cells (to change its color to normal and then back to abnormal) and then clicks the button labeled Solve. This invokes method `countCells` with the coordinates of the last button clicked as its arguments. Figure 5.16 shows a sample grid of buttons with the `x, y` coordinate of each button shown as the button label. The background cells are white, and the abnormal cells are gray. Invoking `countCells` with a starting point of (`x = 4, y = 1`) should return a count of 7. Figure 5.17 shows the blob cells in the temporary color (blue) after the execution of method `countCells`.

When you test this program, make sure you verify that it works for the following cases:

- A starting cell that is on the edge of the grid.
- A starting cell that has no neighboring abnormal cells.
- A starting cell whose only abnormal neighbor cells are diagonally connected to it.
- A "bull's-eye": a starting cell whose neighbors are all normal but their neighbors are abnormal.
- A starting cell that is normal.
- A grid that contains all abnormal cells.
- A grid that contains all normal cells.

FIGURE 5.16
A Sample Grid for Counting Cells in a Blob

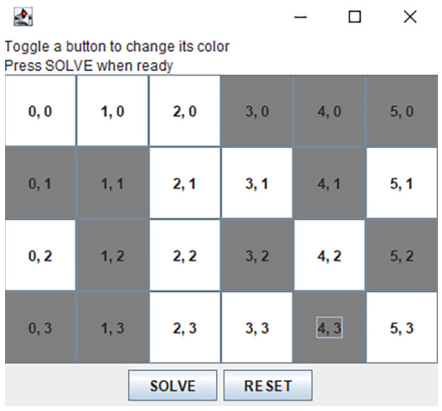


FIGURE 5.17
Blob Cells (in blue) after Execution of `countCells`

EXERCISES FOR SECTION 5.5

SELF-CHECK

1. What is the big-O for the Towers of Hanoi as a function of *n*, where *n* represents the number of disks? Compare it to the function 2^n.
2. How many moves would be required to solve the five-disk problem?
3. Provide a "trace" of the solution to a four-disk problem by showing all the calls to `showMoves` that would be generated.
4. Explain why the first condition of method `countCells` must precede the second condition.

PROGRAMMING

1. Modify method `countCells`, assuming that cells must have a common side in order to be counted in the same blob. This means that they must be connected horizontally or vertically but not diagonally. Under this condition, the value of the method call `aBlob.countCells(4, 1)` would be 4 for the grid in Figure 5.16.
2. Write a method `Blob.restore` that restores the grid to its original state. You will need to reset the color of each cell that is in the temporary color back to its original color.

5.6 Backtracking

In this section, we consider the problem-solving technique called *backtracking*. Backtracking is an approach to implementing systematic trial and error in a search for a solution. An application of backtracking is finding a path through a maze.

If you are attempting to walk through a maze, you will probably follow the general approach of walking down a path as far as you can go. Eventually either you will reach your destination and exit the maze, or you won't be able to go any further. If you exit the maze, you are done. Otherwise, you need to retrace your steps (backtrack) until you reach a fork in the path. At each fork, if there is a branch you did not follow, you will follow that branch hoping to reach your destination. If not, you will retrace your steps again, and so on.

What makes backtracking different from random trial and error is that backtracking provides a systematic approach to trying alternative paths and eliminating them if they don't work out. You will never try the exact same path more than once, and you will eventually find a solution path if one exists.

Problems that are solved by backtracking can be described as a set of choices made by some method. If, at some point, it turns out that a solution is not possible with the current set of choices, the most recent choice is identified and removed. If there is an untried alternative choice, it is added to the set of choices and search continues. If there is no untried alternative choice, then the next most recent choice is removed and an alternative is sought for it. This process continues until either we reach a choice with an untried alternative and can continue our search for a solution, or we determine that there are no more alternative choices to try.

Recursion allows us to implement backtracking in a relatively straightforward manner because we can use each activation frame to remember the choice that was made at that particular decision point.

We will show how to use backtracking to find a path through a maze, but it can be applied to many other kinds of problems that involve a search for a solution. For example, a program

that plays chess may use a kind of backtracking. If a sequence of moves it is considering does not lead to a favorable position, it will backtrack and try another sequence.

CASE STUDY Finding a Path through a Maze

Problem Use backtracking to find and display the path through a maze. From each point in a maze, you can move to the next cell in the horizontal or vertical direction, if that cell is not blocked. So there are at most four possible moves from each point.

Analysis Our maze will consist of a grid of colored cells like the grid used in the previous case study. The starting point is the cell at the top left corner (0, 0), and the exit point is the cell at the bottom right corner (`getNCols() - 1, getNRows() - 1`). All cells that can be part of a path will be in the `BACKGROUND` color. All cells that represent barriers and cannot be part of a path will be in the `ABNORMAL` color. To keep track of a cell that we have visited, we will set it to the `TEMPORARY` color. If we find a path, all cells on the path will be reset to the `PATH` color (a new color for a button defined in `GridColors`). So there are a total of four possible colors for a cell.

Design The following recursive algorithm returns **true** if a path is found. It changes the color of all cells that are visited but found not to be on the path to the temporary color. In the recursive algorithm, each cell `(x, y)` being tested is reachable from the starting point. We can use recursion to simplify the problem of finding a path from cell `(x, y)` to the exit. We know that we can reach any unblocked neighbor cell that is in the horizontal or vertical direction from cell `(x, y)`. So a path exists from cell `(x, y)` to the maze exit if there is a path from a neighbor cell of `(x, y)` to the maze exit. If there is no path from any neighbor cell, we must backtrack and replace `(x, y)` with an alternative that has not yet been tried. That is done automatically through recursion. If there is a path, it will eventually be found and `findMazePath` will return **true**.

Recursive Algorithm for `findMazePath(x, y)`

```
1.  if the current cell is outside the maze
2.      Return false (you are out of bounds).
    else if the current cell is part of the barrier or has already been visited
3.      Return false (you are off the path or in a cycle).
    else if the current cell is the maze exit
4.      Recolor it to the path color and return true (you have successfully completed
        the maze).
    else // Try to find a path from the current path to the exit:
5.      Mark the current cell as on the path by recoloring it to the path color.
6.      for each neighbor of the current cell
7.          if a path exists from the neighbor to the maze exit
8.              Return true.
        // No neighbor of the current cell is on the path.
9.      Mark the current cell as visited by recoloring it to the temporary color and
        return false.
```

If no stopping case is reached (Steps 2, 3, or 4), the recursive case (the **else** clause) marks the current cell as being on the path and then tests whether there is a path from any neighbor of the current cell to the exit. If a path is found, we return **true** and begin unwinding from the recursion. During the process of unwinding from the recursion, the method will continue to return **true**. However, if all neighbors of the current cell are tested without finding a path, this means that the current cell cannot be on the path, so we recolor it to the temporary color and return **false** (Step 9). Next, we backtrack to a previous call and try to find a path through a cell that is an alternative to the cell just tested. The cell just tested will have been marked as visited (the temporary color), so we won't try using it again.

Note that there is no attempt to find the shortest path through the maze. We just show the first path that is found.

Implementation

Listing 5.4 shows class Maze with data field maze (type `TwoDimGrid`). There is a wrapper method that calls recursive method `findMazePath` with its argument values set to the coordinates of the starting point (0, 0). The wrapper method returns the result of this call (**true** or **false**).

The recursive version of `findMazePath` begins with three stopping cases: two unsuccessful and one successful [(x, y) is the exit point]. The recursive case contains an **if** condition with four recursive calls. Because of short-circuit evaluation, if any call returns **true**, the rest are not executed. The arguments for each call are the coordinates of a neighbor cell. If a path exists from a neighbor to the maze exit, then the neighbor is part of the solution path, so we return **true**. If a neighbor cell is not on the solution path, we try the next neighbor until all four neighbors have been tested. If there is no path from any neighbor, we recolor the current cell to the temporary color and return **false**.

LISTIN 5.4
Class Maze

```
import java.awt.*;
import static GridColors.*;

/** Class that solves maze problems with backtracking. */
public class Maze {

    /** The maze */
    private TwoDimGrid maze;

    public Maze(TwoDimGrid m) {
        maze = m;
    }

    /** Wrapper method. */
    public boolean findMazePath() {
        return findMazePath(0, 0); // (0, 0) is the start point.
    }
    /** Attempts to find a path through point (x, y).
        pre: Possible path cells are in BACKGROUND color;
             barrier cells are in ABNORMAL color.
        post: If a path is found, all cells on it are set to the
              PATH color; all cells that were visited but are
              not on the path are in the TEMPORARY color.
```

```
        @param x The x-coordinate of current point
        @param y The y-coordinate of current point
        @return If a path through (x, y) is found, true;
                otherwise, false
     */
    public boolean findMazePath(int x, int y) {
        if (x < 0 || y < 0
            || x >= maze.getNCols() || y >= maze.getNRows())
            return false; // Cell is out of bounds.
        else if (!maze.getColor(x, y).equals(BACKGROUND))
            return false; // Cell is on barrier or dead end.
        else if (x == maze.getNCols() - 1
                 && y == maze.getNRows() - 1) {
            maze.recolor(x, y, PATH); // Cell is on path
            return true;              // and is maze exit.
        } else { // Recursive case.
            // Attempt to find a path from each neighbor.
            // Tentatively mark cell as on path.
            maze.recolor(x, y, PATH);
            if (findMazePath(x - 1, y)
                || findMazePath(x + 1, y)
                || findMazePath(x, y - 1)
                || findMazePath(x, y + 1 ) ) {
                return true;
            }
            maze.recolor(x, y, TEMPORARY); // Dead end.
            return false;
        }
    }
}
```

The Effect of Marking a Cell as Visited

If a path can't be found from a neighbor of the current cell to the maze exit, the current cell is considered a "dead end" and is recolored to the temporary color. You may be wondering whether the program would still work if we just recolored it to the background color. The answer is "yes." In this case, cells that turned out to be dead ends or cells that were not visited would be in the background color after the program terminated. This would not affect the ability of the algorithm to find a path or to determine that none exists; however, it would affect the algorithm's efficiency. After backtracking, the method could try to place on the path a cell that had been found to be a dead end. The cell would be classified once again as a dead end. Marking it as a dead end (color `TEMPORARY`) the first time prevents this from happening.

To demonstrate the efficiency of this approach, we tested the program on a maze with four rows and six columns that had a single barrier cell at the maze exit. When we recolored each dead end cell in the `TEMPORARY` color, it took 93 recursive calls to `findMazePath` to determine that a path did not exist. When we recolored each tested cell in the `BACKGROUND` color, it took 177,313 recursive calls to determine that a path did not exist.

Testing

We will use class `TwoDimGrid` and class `MazeTest` (from the textbook Web site) to test the maze. The `MazeTest` class is very similar to `BlobTest`. The main method prompts for the grid dimensions and creates a new `TwoDimGrid` object with those dimensions. The class constructor builds the graphical user interface (GUI) for the maze solver, including the button panel, and registers a listener for each button.

Initially, all cells are in the background color (white). To set up the maze, the user toggles each cell which is a barrier cell, changing its color from white to gray. When the SOLVE button is clicked, method `MazeTest.actionPerformed` calls `findMazePath` and displays its result.

Figure 5.18 shows the GUI before the SOLVE button is clicked. Figure 5.19 shows it after the SOLVE button is clicked and the final path is displayed in blue (`PATH` color). The cells that were visited but then rejected (not on the path) are in black (`TEMPORARY` color). The cells that were not visited remain white.

You should test this with a variety of mazes, some that can be solved and some that can't (no path exists). You should also try a maze that has no barrier cells and one that has a single barrier cell at the exit point. In the latter case, no path exists.

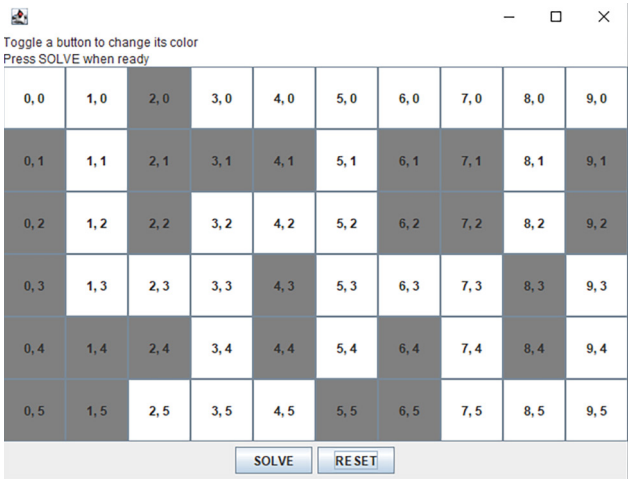


FIGURE 5.18
Maze as Grid of Buttons before SOLVE Is Clicked

FIGURE 5.19
Maze as Grid of Buttons after SOLVE Is Clicked. The path found is in blue.

EXERCISES FOR SECTION 5.6

SELF-CHECK

1. The terminating conditions in `findMazePath` must be performed in the order specified. What could happen if the second or third condition was evaluated before the first? If the third condition was evaluated before the second condition?
2. Does it matter in which order the neighbor cells are tested in `findMazePath`? How could this order affect the path that is found?
3. Is the path shown in Figure 5.19 the shortest path to the exit? If not, list the cells on the shortest path.

PROGRAMMING

1. Show the interface `GridColors`.
2. Write a `Maze.resetTemp` method that recolors the cells that are in the `TEMPORARY` color to the `BACKGROUND` color.
3. Write a `Maze.restore` method that restores the maze to its initial state.

Chapter Review

- A recursive method has the following form, where Step 2 is the base case, and Steps 3 and 4 are the recursive case:

  1. **`if`** the problem can be solved for the current value of *n*
  2. Solve it.

  `else`

  3. Recursively apply the algorithm to one or more problems involving smaller values of *n*.
  4. Combine the solutions to the smaller problems to get the solution to the original.

- To prove that a recursive algorithm is correct, you must
  - — Verify that the base case is recognized and solved correctly.
  - — Verify that each recursive case makes progress toward the base case.
  - — Verify that if all smaller problems are solved correctly, then the original problem must also be solved correctly.
- The run-time stack uses activation frames to keep track of argument values and return points during recursive method calls. Activation frames can be used to trace the execution of a sequence of recursive method calls.
- Mathematical sequences and formulas that are defined recursively can be implemented naturally as recursive methods.

- Recursive data structures are data structures that have a component that is the same data structure. A linked list can be considered a recursive data structure because each node consists of a data field and a reference to a linked list. Recursion can make it easier to write methods that process a linked list.
- Two problems that can be solved using recursion were investigated: the Towers of Hanoi problem and counting cells in a blob.
- Backtracking is a technique that enables you to write programs that can be used to explore different alternative paths in a search for a solution.

User-Defined Classes in This Chapter

```
Blob
BlobTest
GridColors
LinkedListRec
Maze
MazeTest
RecursiveMethods
TowersOfHanoi
TwoDimGrid
```

Quick-Check Exercises

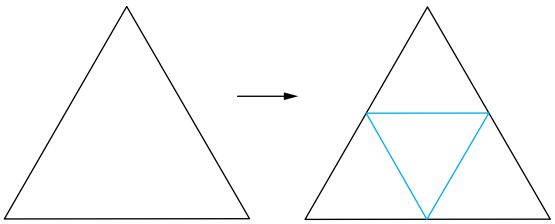
1. A recursive method has two cases:_____ and_____.
2. Each recursive call of a recursive method must lead to a situation that is _____ to the _____ case.
3. The control statement used in a recursive method is the_____ statement.
4. What three things are stored in an activation frame? Where are the activation frames stored?
5. You can sometimes substitute_____ for recursion.
6. Explain how a recursive method might cause a stack overflow exception.
7. If you have a recursive method and an iterative method that calculate the same result, which do you think would be more efficient? Explain your answer.
8. Binary search is an O(____) algorithm and linear search is an O(____) algorithm.
9. Towers of Hanoi is an O(_____) algorithm. Explain your answer.
10. Why did you need to provide a wrapper method for recursive methods `linearSearch` and `binarySearch`?
11. Why did you need to provide a wrapper method for recursive methods in the `LinkedListRec` class?

Review Questions

1. Explain the use of the run-time stack and activation frames in processing recursive method calls.
2. What is a recursive data structure? Give an example.
3. For class `LinkedListRec`, write a recursive search method that returns **true** if its target argument is found and **false** otherwise. If you need a wrapper method, provide one.
4. For class `LinkedListRec`, write a recursive `replaceFirst` method that replaces the first occurrence of a reference to its first argument with a reference to its second argument. If you need a wrapper method, provide one.
5. For Towers of Hanoi, show the output string that would be created by the method call `showMoves(3, 'R', 'M', 'L')`. Also, show the sequence of method calls.
6. For the counting cells in a blob problem, show the activation frames in the first 10 recursive calls to `countCells` following `countCells(4, 1)`.
7. For the maze path found in Figure 5.19, explain why cells (3, 4), (2, 5), (3, 5), and (4, 5) were never visited and why cells (5, 1) and (3, 0) through (9, 0) were visited and rejected. Show the activation frames for the first 10 recursive calls in solving the maze.

Programming Projects

1. Download and run class `BlobTest`. Try running it with a data file made up of lines consisting of 0s and 1s with no spaces between them. Also run it without a data file.
2. Download and run class `MazeTest`. Try running it with a data file made up of lines consisting of 0s and 1s with no spaces between them. Also run it without a data file.
3. Write a recursive method that converts an integer to a binary string. Write a recursive method that converts an integer to a decimal string.
4. Write a `LinkedListRec` class that has the following methods: `size`, isEmpty, `insertBefore`, `insertAfter`, `addAtHead`, `addAtEnd`, `remove`, `replace`, `peekFront`, `peekEnd`, `removeFront`, `removeEnd`, `toString`. Use recursion to implement most of these methods.
5. As discussed in Chapter 3, a palindrome is a word that reads the same left to right as right to left. Write a recursive method that determines whether its argument string is a palindrome.
6. Write a program that will read a list of numbers and a desired sum, then determine the subset of numbers in the list that yield that sum if such a subset exists.
7. Write a recursive method that will dispense change for a given amount of money. The method will display all combinations of quarters, dimes, nickels, and pennies that equal the desired amount.
8. Produce the Sierpinski fractal. Start by drawing an equilateral triangle that faces upward. Then draw an equilateral triangle inside it that faces downward.



   Continue this process on each of the smaller triangles, but not the center one. Stop when the side dimension for a triangle to be drawn is smaller than a specified minimum size.
9. Write a recursive method for placing eight queens on a chessboard. The eight queens should be placed so that no queen can capture another. Recall that a queen can move in the horizontal, vertical, or diagonal direction.
10. Write a recursive method that will find and list all of the one-element sequences of a letters in a `char[]` array, then all the two-element sequences, then all of the three element sequences, and so on such that the characters in each sequence appear in the same order as they are in the array. For example, for the following array:

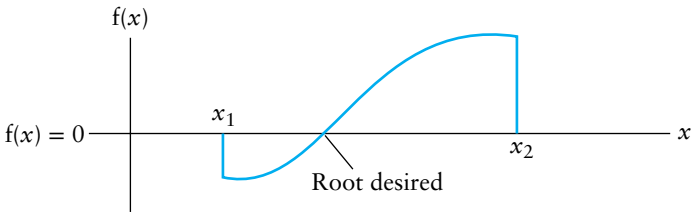
    ```
    char[] letters = {'A', 'C', 'E', 'G'};
    ```
    the one-element sequences are "A", "C", "E", and "G"

    the two-element sequences are "AC", "AE", "AG", "CE", "CG", "EG"

    the three-element sequences are "ACE", "ACG", "AEG", and "CEG"

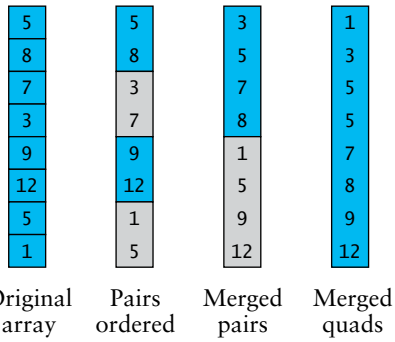
    the four-element sequence is "ACEG"
11. One method of solving a continuous numerical function for a root implements a technique similar to the binary search. Given a numerical function, defined as f(x), and two values of x that are known to bracket one of the roots, an approximation to this root can be determined through a method of repeated division of this bracket. For a set of values of x to bracket a root, the value of the function for one x must be negative and the other must be positive as illustrated below, which plots f(x) for values of x between $x1$ and $x2$.

    The algorithm requires that the midpoint between $x1$ and $x2$ be evaluated in the function, and if it equals zero the root is found; otherwise, $x1$ or $x2$ is set to this midpoint. To determine whether to replace $x1$ or $x2$, the sign of the midpoint is compared against the signs of the values f($x1$) and f($x2$). The midpoint replaces the x ($x1$ or $x2$) whose function value has the same sign as the function value at the midpoint.



This algorithm can be written recursively. The terminating conditions are true when either the midpoint evaluated in the function is zero or the absolute value of $x1 - x2$ is less than some small predetermined value (e.g., 0.0005). If the second condition occurs, then the root is said to be approximately equal to the last midpoint.

12. We can use a merge technique to sort two arrays. The *mergesort* begins by taking adjacent pairs of array values and ordering the values in each pair. It then forms groups of four elements by merging adjacent pairs (first pair with second pair, third pair with fourth pair, etc.) into another array. It then takes adjacent groups of four elements from this new array and merges them back into the original array as groups of eight, and so on. The process terminates when a single group is formed that has the same number of elements as the array. The mergesort is illustrated here for an array with eight elements. Write a `mergeSort` method.



Answers to Quick-Check Exercises

1. A recursive method has two cases: *base case* and *recursive case*.
2. Each recursive call of a recursive method must lead to a situation that is *closer* to the *base* case.
3. The control statement used in a recursive method is the **`if`** statement.
4. An activation frame stores the following information on the run-time stack: the method argument values, the method local variable values, and the address of the return point in the caller of the method.
5. You can sometimes substitute *iteration* for recursion.
6. A recursive method that doesn't stop would continue to call itself, eventually pushing so many activation frames onto the run-time stack that a stack overflow exception would occur.
7. An *iterative* method would generally be more efficient because there is more overhead associated with multiple method calls.
8. Binary search is an $O(\log_2 n)$ algorithm and linear search is an $O(n)$ algorithm.
9. Towers of Hanoi is an $O(2^n)$ algorithm because each problem splits into two problems at the next lower level.
10. Both search methods should be called with the array name and target as arguments. However, the recursive linear search method needs the subscript of the element to be compared to the target. The binary search method needs the search array bounds.
11. The wrapper method should be applied to a `LinkedListRec` object. The recursive method needs the current list head as an argument.

Chapter 6

Trees

Chapter Objectives

- To learn how to use a tree to represent a hierarchical organization of information
- To learn how to use recursion to process trees
- To understand the different ways of traversing a tree
- To understand the difference between binary trees, binary search trees, and heaps
- To learn how to implement binary trees, binary search trees, and heaps using linked data structures and arrays
- To learn how to use Lambda Expressions and Functional Interfaces to simplify coding
- To learn how to use a binary search tree to store information so that it can be retrieved an efficient manner
- To learn how to use a Huffman tree to encode characters using fewer bits than ASCII or Unicode, resulting in smaller files and reduced storage requirements

The data organizations you have studied so far are linear in that each element has only one predecessor or successor. Accessing all the elements in sequence is an O(n) process. In this chapter, we begin our discussion of a data organization that is nonlinear or hierarchical: the tree. Instead of having just one successor like a node in a list, a node in a tree can have multiple successors, but it has just one predecessor. A tree in computer science is like a natural tree, which has a single trunk that may split off into two or more main branches. The predecessor of each main branch is the trunk. Each main branch may spawn several secondary branches (successors of the main branches). The predecessor of each secondary branch is a main branch. In computer science, we draw a tree from the top down, so the root of the tree is at the top of the diagram instead of the bottom.

Because trees have a hierarchical structure, we use them to represent hierarchical organizations of information, such as a class hierarchy, a disk directory and its subdirectories (see Figure 6.1), or a family tree. You will see that trees are recursive data structures because they can be defined recursively. For this reason, many of the methods used to process trees are written as recursive methods.

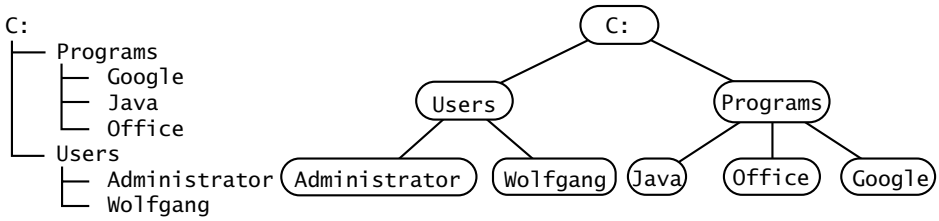


FIGURE 6.1
A Tree Representation of a Disk Directory

This chapter focuses on a restricted tree structure, a binary tree, in which each element has, at most, two successors. You will learn how to use linked data structures and arrays to represent binary trees. You will also learn how to use a special kind of binary tree called a binary search tree to store information (e.g., the words in a dictionary) in an ordered way. Because each element of a binary tree can have two successors, you will see that searching for an item stored in a binary search tree is much more efficient than searching for an item in a linear data structure: (generally $O(\log n)$ for a binary tree versus $O(n)$ for a list).

You also will learn about other kinds of binary trees. Expression trees are used to represent arithmetic expressions. The heap is an ordered tree structure that is used as the basis for a very efficient sorting algorithm and for a special kind of queue called the priority queue. The Huffman tree is used for encoding information and compressing files.

Trees

6.1 Tree Terminology and Applications

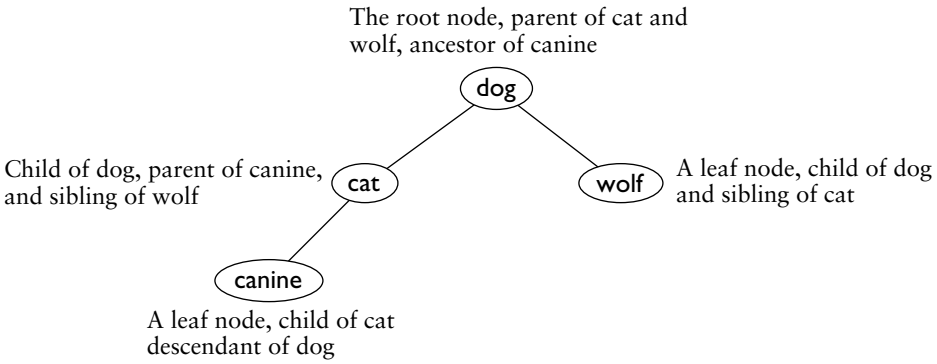
Tree Terminology

We use the same terminology to describe trees in computer science as we do trees in nature. A computer science tree consists of a collection of elements or nodes, with each node linked to its successors. The node at the top of a tree is called its *root* because computer science trees grow from the top down. The links from a node to its successors are called *branches*. The successors of a node are called its *children*. The predecessor of a node is called its *parent*. Each node in a tree has exactly one parent except for the root node, which has no parent. Nodes that have the same parent are *siblings*. A node that has no children is a *leaf node*. Leaf nodes are also known as *external* nodes, and non-leaf nodes are known as *internal* nodes.

A generalization of the parent–child relationship is the *ancestor–descendant relationship*. If node A is the parent of node B, which is the parent of node C, node A is node C's *ancestor*, and node C is node A's *descendant*. Sometimes we say that node A and node C are a grandparent and grandchild, respectively. The root node is an ancestor of every other node in a tree, and every other node in a tree is a descendant of the root node.

Figure 6.2 illustrates these features in a tree that stores a collection of words. The branches are the lines connecting a parent to its children. In discussing this tree, we will refer to a node by the string that it stores. For example, we will refer to the node that stores the string `"dog"` as node *dog*.

FIGURE 6.2
A Tree of Words



A *subtree of a node* is a tree whose root is a child of that node. For example, the nodes *cat* and *canine* and the branch connecting them are a subtree of node *dog*. The other subtree of node *dog* is the tree consisting of the single node *wolf*. The subtree consisting of the single node *canine* is a subtree of node *cat*.

The *level of a node* is a measure of its distance from the root. It is defined recursively as follows:

- If node *n* is the root of tree T, its level is 1.
- If node *n* is not the root of tree T, its level is 1 + the level of its parent.

For the tree in Figure 6.2, node *dog* is at level 1, nodes *cat* and *wolf* are at level 2, and node *canine* is at level 3. Since nodes are below the root, we sometimes use the term *depth* as an alternative term for level. The two have the same meaning.

The *height of a tree* is the number of nodes in the longest path from the root node to a leaf node. The height of the tree in Figure 6.2 is 3 (the longest path goes through the nodes *dog*, *cat*, and *canine*). Another way of saying this is as follows:

- If T is empty, its height is 0.
- If T is not empty, its height is the maximum depth of its nodes.

An alternative definition of the height of a tree is the number of branches in the longest path from the root node to a leaf node + 1.

Binary Trees

The tree in Figure 6.2 is a *binary tree*. Informally, this is a binary tree because each node has at most two subtrees. A more formal definition for a binary tree follows.

A set of nodes T is a binary tree if either of the following is true:

- T is empty.
- If T is not empty, its root node has two subtrees, T_L and T_R, such that T_L and T_R are binary trees.

We refer to T_L as the left subtree and T_R as the right subtree. For the tree in Figure 6.2, the right subtree of node *cat* is empty. The leaf nodes (*wolf* and *canine*) have empty left and right subtrees. This is illustrated in Figure 6.3, where the empty subtrees are indicated by the squares. Generally, the empty subtrees are represented by **`null`** references, but another value may be chosen. From now on, we will consistently use a **`null`** reference and will not draw the squares for the empty subtrees.

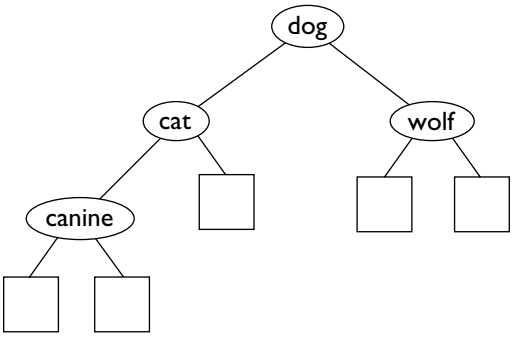


FIGURE 6.3
A Tree of Words with `null` Subtrees Indicated

Some Types of Binary Trees

Next, we discuss three different types of binary trees that are common in computer science.

An Expression Tree

Figure 6.4 shows a binary tree that stores an expression. Each node contains an operator (`+`, `-`, `*`, `/`, `%`) or an operand. The expression in Figure 6.4 corresponds to `(x + y) * ((a + b) / c)`. Operands are stored in leaf nodes. Parentheses are not stored in the tree because the tree structure dictates the order of operator evaluation. Operators in nodes at higher levels are evaluated after operators in nodes at lower levels, so the operator * in the root node is evaluated last. If a node contains a binary operator, its left subtree represents the operator's left operand and its right subtree represents the operator's right operand. The left subtree of the root represents the expression `x + y`, and the right subtree of the root represents the expression `(a + b) / c`.

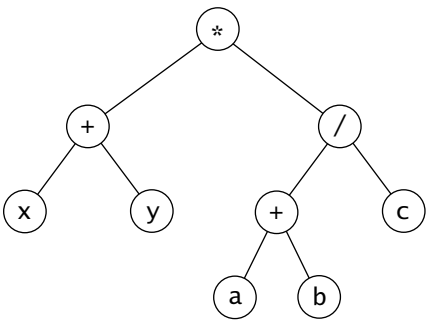


FIGURE 6.4
Expression Tree

A Huffman Tree

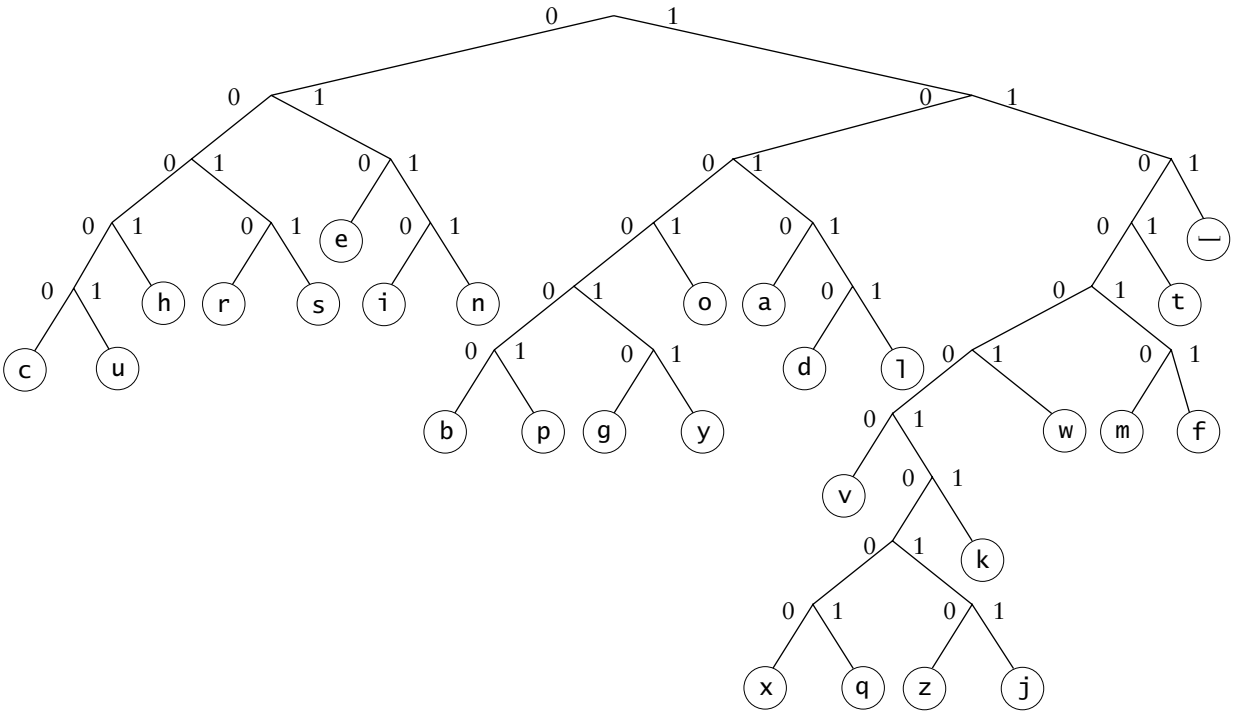
Another use of a binary tree is to represent *Huffman codes* for characters that might appear in a text file. Unlike ASCII or Unicode encoding, which use the same number of bits to encode each character, a Huffman code uses different numbers of bits to encode the letters. It uses fewer bits for the more common letters (e.g., space, *e*, *a*, and *t*) and more bits for the less common letters (e.g., *q*, *x*, and *z*). On average, using Huffman codes to encode text files should give you files with fewer bits than you would get using other codes. Many programs that compress files use Huffman encoding to generate smaller files in order to save disk space or to reduce the time spent sending the files over the Internet.

Figure 6.5 shows the Huffman encoding tree for an alphabet consisting of the lowercase letters and the space character. All the characters are at leaf nodes. The data stored at non-leaf nodes is not shown. To determine the code for a letter, you form a binary string by tracing the path from the root node to that letter. Each time you go left, append a 0, and each time you go right, append a 1. To reach the space character, you go right three times, so the code is 111. The code for the letter *d* is 10110 (right, left, right, right, left).

The two characters shown at level 4 of the tree (space, *e*) are the most common and, therefore, have the shortest codes (111, 010). The next most common characters (*a*, *o*, *i*, etc.) are at level 5 of the tree.

You can store the code for each letter in an array. For example, the code for the space `' '` would be at position 0, the letter `'a'` would be at position 1, and the code for letter `'z'` would be at position 26. You can *encode* each letter in a file by looking up its code in the array.

FIGURE 6.5
Huffman Code Tree



However, to *decode* a file of letters and spaces, you walk down the Huffman tree, starting at the root, until you reach a letter and then append that letter to the output text. Once you have reached a letter, go back to the root. Here is an example. The substrings that represent the individual letters are shown in alternate colors to help you follow the process. The underscore under the substring 111 represents a space character (code is 111).

```
100010100111101010101000101011101000 11
  g    o  _  e  a    g     l   e  s
```

Huffman trees are discussed further in Section 6.7.

A Binary Search Tree

The tree in Figure 6.2 is a *binary search tree* because, for each node, all words in its left subtree precede the word in that node, and all words in its right subtree follow the word in that node. For example, for the root node *dog*, all words in its left subtree (*cat*, *canine*) precede dog in the dictionary, and all words in its right subtree (*wolf*) follow *dog*. Similarly, for the node *cat*, the word in its left subtree (*canine*) precedes it. There are no duplicate entries in a binary search tree.

More formally, we define a binary search tree as follows:

A set of nodes T is a binary search tree if either of the following is true:

- T is empty.
- If T is not empty, its root node has two subtrees, T_L and T_R, such that T_L and T_R are binary search trees and the value in the root node of T is greater than all values in T_L and is less than all values in T_R.

The order relations in a binary search tree expedite searching the tree. A recursive algorithm for searching a binary search tree follows:

Recursive Algorithm for Insertion in a Binary Search Tree

1. **`if`** the tree is empty
2. Return **`null`** *(target is not found)*.

   `else if` the target matches the root node's data
3. Return the data stored at the root node.

   `else if` the target is less than the root node's data
4. Return the result of searching the left subtree of the root.

   `else`
5. Return the result of searching the right subtree of the root.

The first two cases are base cases and self-explanatory. In the first recursive case, if the target is less than the root node's data, we search only the left subtree (T_L) because all data items in T_R are larger than the root node's data and, therefore, larger than the target. Likewise, we execute the second recursive step (search the right subtree) if the target is greater than the root node's data.

Just as with a binary search of an array, each probe into the binary search tree has the potential of eliminating half the elements in the tree. If the binary search tree is relatively balanced (i.e., the depths of the leaves are approximately the same), searching a binary search tree is an $O(\log n)$ process, just like a binary search of an ordered array.

What is the advantage of using a binary search tree instead of just storing elements in an array and then sorting it? A binary search tree never has to be sorted because its elements always satisfy the required order relations. When new elements are inserted (or removed), the

binary search tree property can be maintained. In contrast, an array must be expanded whenever new elements are added, and it must be compacted whenever elements are removed. Both expanding and contracting involve shifting items and are thus O(n) operations.

Full, Perfect, and Complete Binary Trees

The tree on the left in Figure 6.6 is called a *full binary tree* because all nodes have either 2 children or 0 children (the leaf nodes). The tree in the middle is a *perfect binary tree*, which is defined as a full binary tree of height n (n is 3) with exactly $2^n - 1$ (7) nodes. The tree on the right is a *complete binary tree*, which is a perfect binary tree through level $n - 1$ with some extra leaf nodes at level n (the tree height), all toward the left.

General Trees

A general tree is a tree that does not have the restriction that each node of a tree has at most two subtrees. So nodes in a general tree can have any number of subtrees. Figure 6.7 shows a general tree that represents a family tree showing the descendants of King William I (the Conqueror) of England.

FIGURE 6.6 Full, Perfect, and Complete Binary Trees

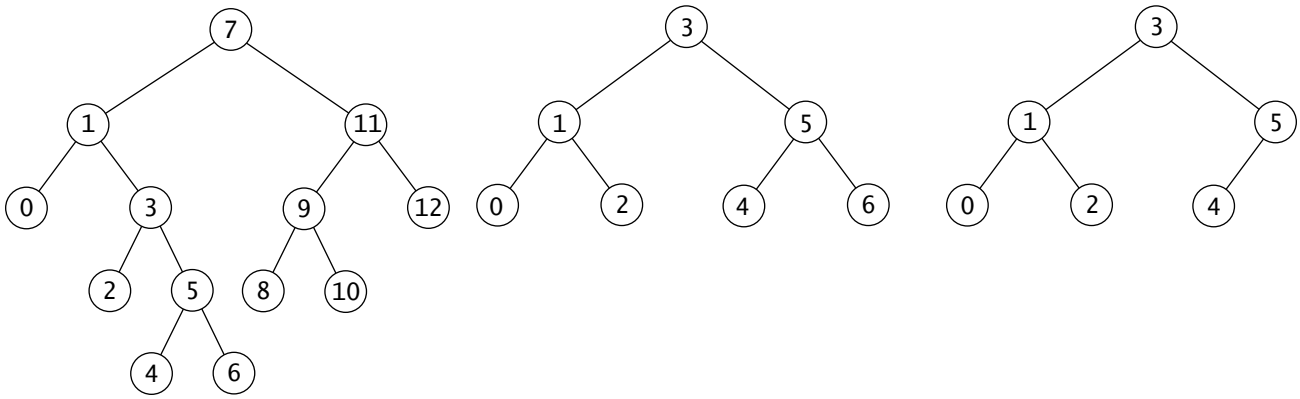
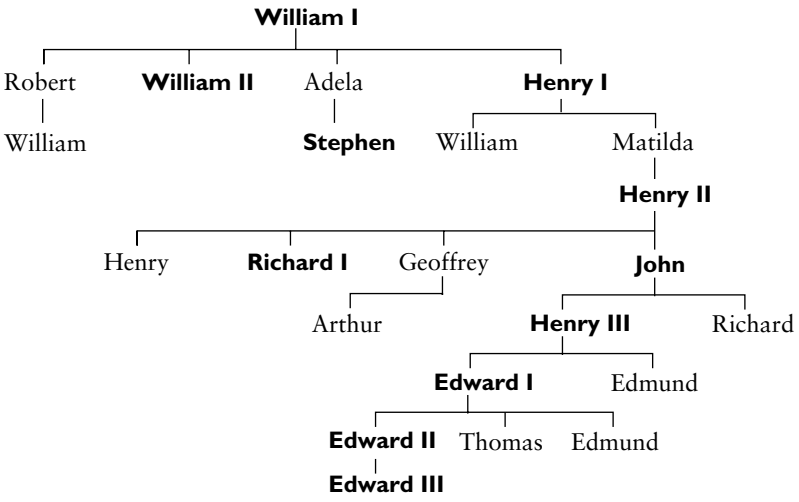


FIGURE 6.7 Family Tree for the Descendants of William I of England



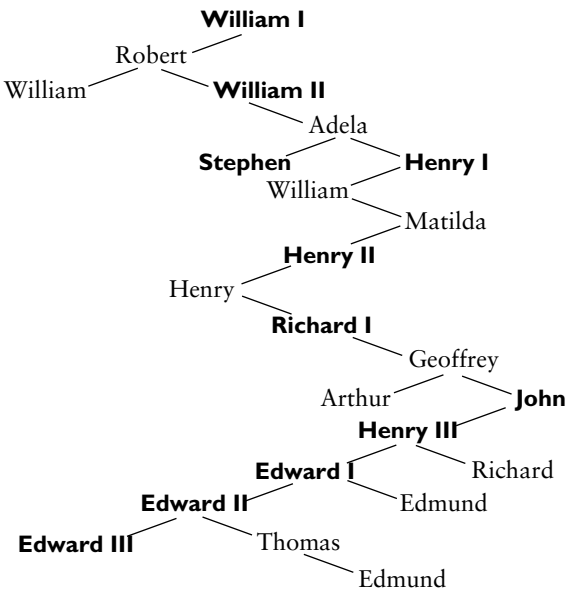


FIGURE 6.8
Binary Tree Equivalent of King Williams Family Tree

We will not discuss general trees in this chapter. However, it is worth mentioning that a general tree can be represented using a binary tree. Figure 6.8 shows a binary tree representation of the family tree in Figure 6.7. We obtained it by connecting the left branch from a node to the oldest child (if any). Each right branch from a node is connected to the next younger sibling (if any).

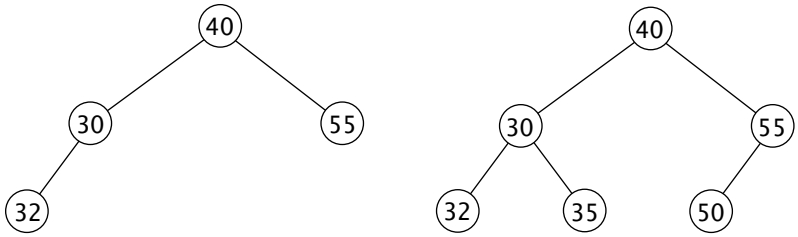
The names of the men who became kings are in boldface type. You would expect the eldest son to succeed his father as king; however, this would not be the case if the eldest male died before his father. For example, Robert died before William I, so William II became king instead. Starting with King John (near the bottom of the tree), the eldest son of each king did become the King of England.

EXERCISES FOR SECTION 6.1

SELF-CHECK

1. Draw binary expression trees for the following infix expressions. Your trees should enforce the Java rules for operator evaluation (higher-precedence operators before lower-precedence operators and left associativity).
   a. `x / y + a - b * c`
   b. `(x * a) - y / b * (c + d)`
   c. `(x + (a * (b - c))/d`
2. Using the Huffman tree in Figure 6.5,
   a. Write the binary string for the message "scissors cuts paper."
   b. Decode the following binary string using the tree in Figure 6.5:
   `11000100010100010010111011000111111100011010101111101101001`

3. For each tree shown below, answer these questions. What is its height? Is it a full tree? Is it a complete tree? Is it a binary search tree? If not, make it a binary search tree.



4. For the binary trees in Figures 6.2–6.5, indicate whether each tree is full, perfect, complete, or none of the above.

5. Represent the general tree in Figure 6.1 as a binary tree.

6.2 Tree Traversals

Often, we want to determine the nodes of a tree and their relationship. We can do this by walking through the tree in a prescribed order and visiting the nodes (processing the information in the nodes) as they are encountered. This process is known as *tree traversal*. We will discuss three kinds of traversal in this section: inorder, preorder, and postorder. These three methods are characterized by when they visit a node in relation to the nodes in its subtrees (T_L and T_R).

- Preorder: Visit root node, traverse T_L, and traverse T_R.
- Inorder: Traverse T_L, visit root node, and traverse T_R.
- Postorder Traverse T_L, traverse T_R, and visit root node.

Because trees are recursive data structures, we can write similar recursive algorithms for all three techniques. The difference in the algorithms is whether the root is visited before the children are traversed (pre), in between traversing the left and right children (in), or after the children are traversed (post).

Algorithm for Preorder Traversal

1. `if` the tree is empty
2. Return.

   `else`
3. Visit the root.
4. Preorder traverse the left subtree.
5. Preorder traverse the right subtree

Algorithm for Inorder Traversal

1. `if` the tree is empty
2. Return.

   `else`
3. Inorder traverse the left subtree.
4. Visit the root.
5. Inorder traverse the right subtree.

Algorithm for Postorder Traversal

1. `if` the tree is empty
2. Return.

   `else`
3. Postorder traverse the left subtree.
4. Postorder traverse the right subtree.
5. Visit the root.

Visualizing Tree Traversals

You can visualize a tree traversal by imagining a mouse that walks along the edge of the tree. If the mouse always keeps the tree to the left (from the mouse's point of view), it will trace the route shown in blue around the tree shown in Figure 6.9. This is known as an *Euler tour*.

If we record each node as the mouse first encounters it (indicated by the arrows pointing down in Figure 6.9), we get the following sequence:

a b d g e h c f i j

This is a preorder traversal because the mouse visits each node before traversing its subtrees. The mouse also walks down the left branch (if it exists) of each node before going down the right branch, so the mouse visits a node, traverses its left subtree, and traverses its right subtree.

If we record each node as the mouse returns from traversing its left subtree (indicated by the arrows pointing to the right in Figure 6.9), we get the following sequence:

d g b h e a i f j c

This is an inorder traversal. The mouse traverses the left subtree, visits the root, and then traverses the right subtree. Node *d* is visited first because it has no left subtree.

If we record each node as the mouse last encounters it (indicated by the arrows pointing up in Figure 6.9), we get the following sequence:

g d h e b i j f c a

This is a postorder traversal because we visit the node after traversing both its subtrees. The mouse traverses the left subtree, traverses the right subtree, and then visits the node.

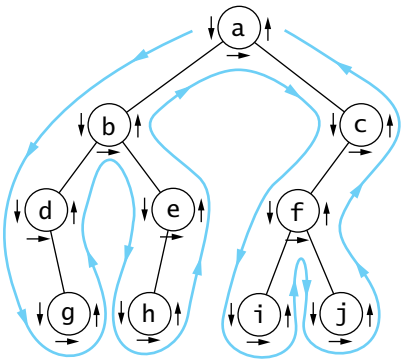


FIGURE 6.9
Traversal of a Binary Tree

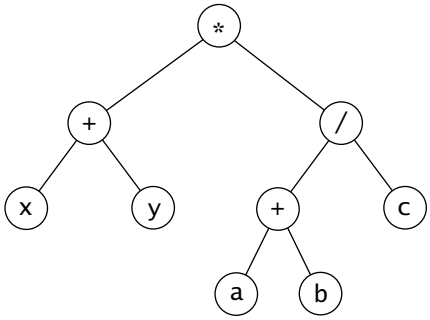
Traversals of Binary Search Trees and Expression Trees

An inorder traversal of a binary search tree results in the nodes being visited in sequence by increasing data value. For example, for the binary search tree shown in Figure 6.2, the inorder traversal would visit the nodes in the sequence:

canine, *cat*, *dog*, *wolf*

Traversals of expression trees give interesting results. If we perform an inorder traversal of the expression tree first shown in Figure 6.4 and repeated here, we visit the nodes in the sequence `x + y* a + b / c`. If we insert parentheses where they belong, we get the infix expression

```
(x + y) * ((a + b) / c)
```



The postorder traversal of this tree would visit the nodes in the sequence

```
x y + a b + c / *
```

which is the postfix form of the expression. To illustrate this, we show the *operand–operand–operator* groupings under the expression.

The preorder traversal visits the nodes in the sequence

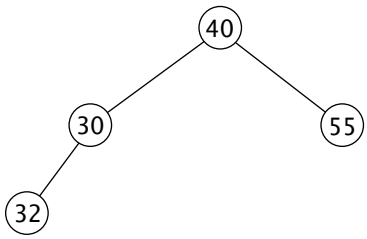
```
* + x y / + a b c
```

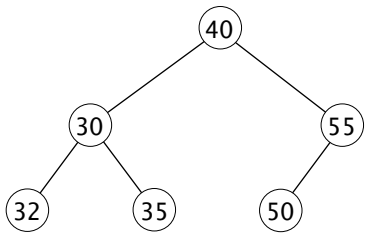
which is the prefix form of the expression. To illustrate this, we show the *operator–operand–operand* groupings under the expression.

EXERCISES FOR SECTION 6.2

SELF-CHECK

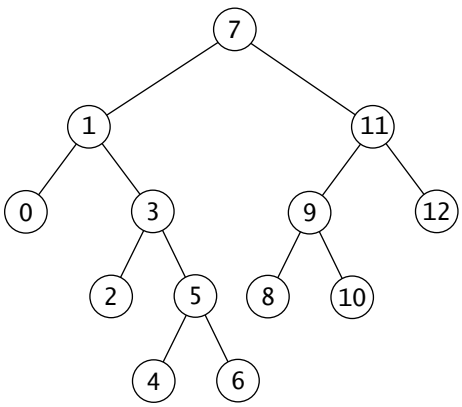
1. For the following trees:

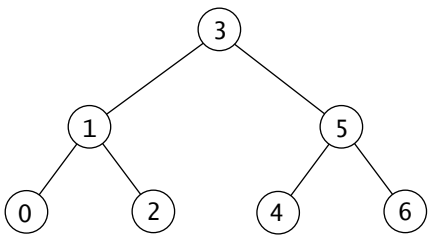


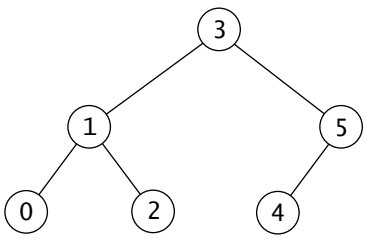


   If visiting a node displays the integer value stored, show the inorder, preorder, and postorder traversal of each tree.

2. Repeat Exercise 1 above for the trees in Figure 6.6, redrawn below.







3. Draw an expression tree corresponding to each of the following:
   a. Inorder traversal is `x / y + 3 * b / c` (Your tree should represent the Java meaning of the expression.)
   b. Postorder traversal is `x y z + a b - c * / -`.
   c. Preorder traversal is `* + a - x y / c d`.
4. Explain why the statement "Your tree should represent the Java meaning of the expression" was not needed for parts b and c of Exercise 3 above.

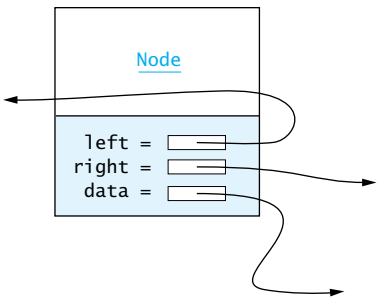
6.3 Implementing a `BinaryTree` Class

In this section, we show how to use linked data structures to represent binary trees and binary tree nodes. We begin by focusing on the structure of a binary tree node.

The `Node<E>` Class

Just as for a linked list, a node consists of a data part and links (references) to successor nodes. So that we can store any kind of data in a tree node, we will make the data part a reference of type `E`. Instead of having a single link (reference) to a successor node as in a list, a binary tree node must have links (references) to both its left and right subtrees. Figure 6.10 shows the structure of a binary tree node; Listing 6.1 shows its implementation.

FIGURE 6.10
Linked Structure to Represent a `Node`



LISTING 6.1
Nested Class `Node`

```
/** Class to encapsulate a tree node. */
protected static class Node<E> {
    // Data Fields
    /** The information stored in this node. */
    protected E data;
    /** Reference to the left child. */
    protected Node<E> left;
    /** Reference to the right child. */
    protected Node<E> right;

    // Constructors
    /** Construct a node with given data and no children.
        @param data The data to store in this node
     */
    public Node(E data) {
        this.data = data;
        left = null;
        right = null;
    }
```

```
    // Methods
    /** Return a string representation of the node.
        @return a string representation of data
     */
    public String toString () {
        return data.toString();
    }
}
```

Class `Node<E>` is nested within class `BinaryTree<E>`. Note that it is declared **`protected`** and its data fields are all **`protected`**. Later, we will use the `BinaryTree<E>` and `Node<E>` classes as superclasses. By declaring the nested `Node<E>` class and its data fields protected, we make them accessible in the subclasses of `BinaryTree<E>` and `Node<E>`.

The constructor for class `Node<E>` creates a leaf node (both left and right are `null`). The `toString` method for the class returns a string representation of the data in the node.

The `BinaryTree<E>` Class

Table 6.1 shows the design of the `BinaryTree<E>` class. The single data field `root` references the root node of a `BinaryTree<E>` object. It has protected visibility because we will need to access it in subclass `BinarySearchTree`, discussed later in this chapter. In Figure 6.11, we draw the expression tree for `((x + y) * (a / b))` using our `Node` representation. Each character shown as tree data would be stored in a `Character` object.

TABLE 6.1
Design of the `BinaryTree<E>` Class

Data Field	Attribute
`protected Node<E> root`	Reference to the root of the tree
Constructor	**Behavior**
`public BinaryTree()`	Constructs an empty binary tree
`protected BinaryTree(Node<E> root)`	Constructs a binary tree with the given node as the root
`public BinaryTree(E data, BinaryTree<E> leftTree, BinaryTree<E> rightTree)`	Constructs a binary tree with the given data at the root and the two given subtrees
Method	**Behavior**
`public BinaryTree<E> getLeftSubtree()`	Returns the left subtree
`public BinaryTree<E> getRightSubtree()`	Returns the right subtree
`public E getData()`	Returns the data in the root
`public boolean isLeaf()`	Returns **`true`** if this tree is a leaf, **`false`** otherwise
`public String toString()`	Returns a `String` representation of the tree
`public void preOrderTraverse(BiConsumer<E, Integer> consumer)`	Performs a preorder traversal of the tree. Each node and its depth are passed to the `consumer` function (covered in the next section)
`public static BinaryTree<E> readBinaryTree(Scanner scan)`	Constructs a binary tree by reading its data using `Scanner scan`

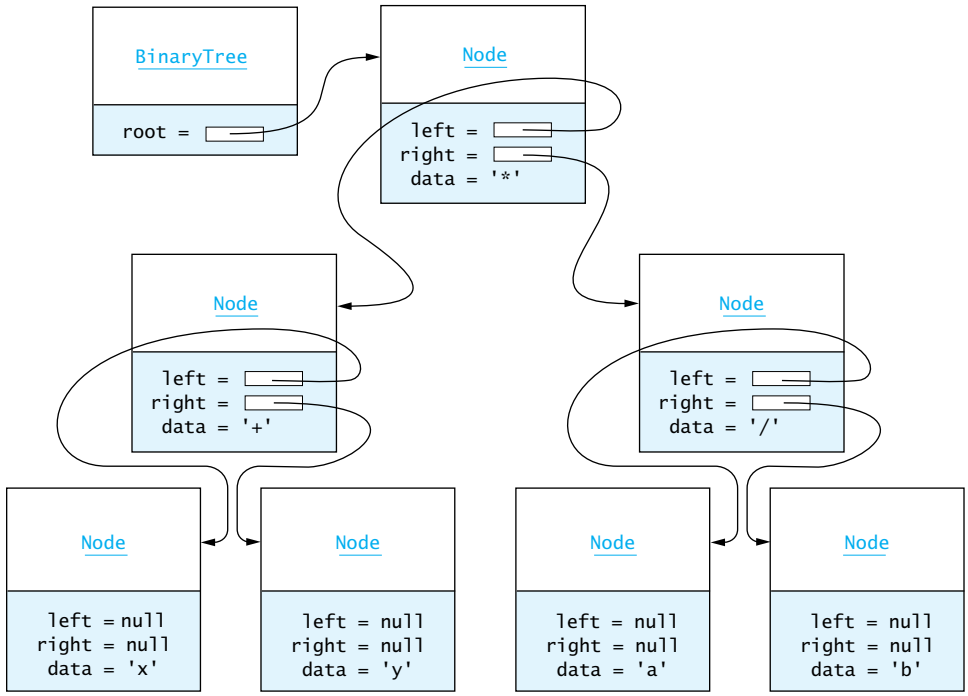


FIGURE 6.11
Linked Representation of Expression Tree `((x + y) * (a / b))`

EXAMPLE 6.1 Assume the tree drawn in Figure 6.11 is referenced by variable `bT` (type `BinaryTree`).

- `bT.root.data` references the `Character` object storing `'*'`.
- `bT.root.left` references the left subtree of the root (the root node of tree `x + y`).
- `bT.root.right` references the right subtree of the root (the root node of tree `a / b`).
- `bT.root.right.data` references the `Character` object storing `'/'`.

The class heading and data field declarations follow:

```
import java.io.*;

/** Class for a binary tree that stores type E objects. */
public class BinaryTree<E> {

    // Insert inner class Node<E> here.

    // Data Field
    /** The root of the binary tree */
    protected Node<E> root;
    . . .
```

The Constructors

There are three constructors: a no-parameter constructor, a constructor that creates a tree with a given node as its root, and a constructor that builds a tree from a data value and two trees.