Programming
and
Data Structures
with
Java and JUnit

Rick Mercer

Chapter	Title
1	Program Development
2	Java Fundamentals
3	Objects and JUnit
4	Methods
5	Selection (if-else)
6	Repetition (while and for loops)
7	Arrays (one subscript)
8	Search and Sort
9	Classes with Instance Variables
10	An Array Instance Variable (a collection class)
11	Two-Dimensional Arrays (two subscripts)
12	Algorithm Analysis
13	Collection Considerations
14	A List ADT
15	The Singly Linked Structure
16	Stacks and Queues
17	Recursion
18	Binary Trees
19	Binary Search Trees

Chapter 1

Program Development

We begin with a need for a computer-based solution to a problem. The need may be expressed in one or two paragraphs as a problem specification. The progression from understanding a problem specification to achieving a working computer-based implementation is known as problem solving. After studying this chapter, you will understand

- one example of problem solving
- the characteristics of an algorithm
- how algorithmic patterns help in program design

There are many approaches to program development. This chapter begins by examining a strategy with these three steps: analysis, design, and implementation.

Phase of Program Development	Activity
Analysis	Understand the problem.
Design	Design an algoritm that outlines a solution
Implementation	Code an executable program ready to be used

Our study of computing fundamentals begins with an example of this particular approach to program development. Each of these three phases will be exemplified with a simple case study—one particular problem—compute a course grade

Analysis *(inquiry, examination, study)*

Program development may begin with a study, or **analysis**, of a problem. Obviously, to determine what a program is to do, you must first understand the problem. If the problem is written down, you can begin the analysis phase by simply reading the problem.

While analyzing the problem, it proves helpful to name the data that represent information. For example, you might be asked to compute the maximum weight allowed for a successful liftoff of a particular airplane from a given runway under certain thrust-affecting weather conditions such as temperature and wind direction. While analyzing the problem specification, you might name the desired information `maximumWeight`. The data required to compute that information could have names such as `temperature` and `windDirection`.

Although such data do not represent the entire solution, they do represent an important piece of the puzzle. The data names are symbols for what the program will need and what the program will compute. One value needed to compute `maximumWeight` might be `19.0` for `temperature`. Such data values must often be manipulated—or processed—in a variety of ways to produce the desired result. Some values must be obtained from the user, other values must be multiplied or added, and still other values must be displayed on the computer screen.

At some point, these data values will be stored in computer memory. The values in the same memory location can change while the program is running. The values also have a type, such as integers or numbers with decimal points (these two different types of values are stored differently in computer memory). These named pieces of memory that store a specific type of value that can change while a program is running are known as **variables**.

You will see that there also are operations for manipulating those values in meaningful ways. It helps to distinguish the data that must be displayed—**output**—from the data required to compute that result—**input**. These named pieces of memory that store values are the variables that summarize what the program must do.

Input and Output

Output: Information the computer must display.

Input: Information a user must supply to solve a problem.

A problem can be better understood by answering this question: What is the output given certain input? Therefore, it is a good idea to provide an example of the problem with pencil and paper. Here are two problems with variable names selected to accurately describe the stored values.

Problem	Data Name	Input or Output	Sample Problem
Compute a monthly loan payment	`amount`	Input	`12500.00`
	`rate`	Input	`0.08`
	`months`	Input	`48`
	`payment`	Output	`303.14`

Problem	Data Name	Input or Output	Sample Problem
Count how often Shakespeare wrote a particular word in a particular play	`aBardsWork`	Input	`Much Ado About Nothing`
	`theWord`	Input	`the`
	`howOften`	Output	`220`

In summary, problems are analyzed by doing these things:

1. Reading and understanding the problem specification.
2. Deciding what data represent the answer—the output.
3. Deciding what data the user must enter to get the answer—the input.
4. Creating a document (like those above) that summarizes the analysis. This document is input for the next phase of program development—design.

In textbook problems, the variable names and type of values (such as integers or numbers with a decimal point) that must be input and output are sometimes provided. If not, they are relatively easy to recognize. In real-world problems of significant scale, a great deal of effort is expended during the analysis phase. The next subsection provides an analysis of a small problem.

Self-Check

1-1 Given the problem of converting British pounds to U.S. dollars, provide a meaningful name for the value that must be input by the user. Give a meaningful name for a value that must be output.

1-2 Given the problem of selecting one CD from a 200-compact-disc player, what name would represent all of the CDs? What name would be appropriate to represent one particular CD selected by the user?

An Example of Analysis

Problem: Using the grade assessment scale to the right, compute a course grade as a weighted average of two tests and one final exam.

Item	**Percentage of Final Grade**
Test 1	25%
Test 2	25%
Final Exam	50%

Analysis begins by reading the problem specification and establishing the desired output and the required input to solve the problem. Determining and naming the output is a good place to start. The output stores the answer to the problem. It provides insight into what the program must do. Once the need for a data value is discovered and given a meaningful name, the focus can shift to what must be accomplished. For this particular problem, the desired output is the actual course grade. The name `courseGrade` represents the requested information to be output to the user.

This problem becomes more generalized when the user enters values to produce the result. If the program asks the user for data, the program can be used later to compute course grades for many students with any set of grades. So let's decide on and create names for the values that must be input. To determine `courseGrade`, three values are required: `test1`, `test2`, and `finalExam`. The first three analysis activities are now complete:

1. Problem understood.
2. Information to be output: `courseGrade`.
3. Data to be input: `test1`, `test2`, and `finalExam`.

However, a sample problem is still missing. Consider these three values

- `test1` ***74.0***
- `test2` ***79.0***
- `finalExam` ***84.0***
- `courseGrade` ***?***

Sample inputs along with the expected output provide an important benefit–we have an expected result for one set of inputs. In this problem, to create this `courseGrade` problem, we must understand the difference between a simple average and a weighted average. Because the three input items comprise different portions of the final grade (either 25% or 50%), the problem involves computing a weighted average. The simple average of the set 74.0, 79.0, and 84.0 is 79.0; each test is measured equally. However, the weighted average computes differently. Recall that `test1` and `test2` are each worth 25%, and `finalExam` weighs in at 50% of the final grade. When `test1` is 74.0, `test2` is 79.0, and `finalExam` is 84.0, the weighted average computes to 80.25.

```
(0.25 x test1) + (0.25 x test2) + (0.50 x finalExam)
 (0.25 x 74.0) +  (0.25 x 79.0) +   (0.50 x 84.0)
     18.50     +      19.75     +       42.00
                      80.25
```

With the same exact grades, the weighted average of 80.25 is different from the simple average (79.0). Failure to follow the problem specification could result in students who receive grades lower, or higher, than they actually deserve.

The problem has now been analyzed, the input and output have been named, it is understood what the computer-based solution is to do, and one sample problem has been given. Here is a summary of analysis:

Problem	Data Name	Input or Output	Sample Problem
Compute a course grade	`test1`	Input	`74.0`
	`test2`	Input	`79.0`
	`finalExam`	Input	`84.0`
	`courseGrade`	Output	`80.25`

The next section presents a method for designing a solution. The emphasis during design is on placing the appropriate activities in the proper order to solve the problem.

Self-Check

1-3 Complete an analysis for the following problem. You will need a calculator to determine output.

Problem: Show the future value of an investment given its present value, the number of periods (years, perhaps), and the interest rate. Be consistent with the interest rate and the number of periods; if the periods are in years, then the annual interest rate must be supplied (0.085 for 8.5%, for example). If the period is in months, the monthly interest rate must be supplied (0.0075 per month for 9% per year, for example). The formula to compute the future value of money is future value = present value * $(1 + \text{rate})^{\text{periods}}$.

1.3 Design *(model, think, plan, devise, pattern, outline)*

Design refers to the set of activities that includes specifying an algorithm for each program component. In later chapters, you will see functions used as the basic building blocks of programs. Later you will see classes used as the basic building blocks of programs. A class is a collection of functions and values. In this chapter, the building block is intentionally constrained to a component known as a **program**. Therefore, the design activity that follows is limited to specifying an algorithm for this program.

An **algorithm** is a step-by-step procedure for solving a problem or accomplishing some end, especially by a computer. A good algorithm must

- list the activities that need to be carried out
- list those activities in the proper order

Consider an algorithm to bake a cake:

1. Preheat the oven
2. Grease the pan
3. Mix the ingredients
4. Pour the ingredients into the pan
5. Place the cake pan in the oven
6. Remove the cake pan from the oven after 35 minutes

If the order of the steps is changed, the cook might get a very hot cake pan with raw cake batter in it. If one of these steps is omitted, the cook probably won't get a baked cake—or there might be a fire. An experienced cook may not need such an algorithm. However, cake-mix marketers cannot and do not presume that their customers have this experience. Good algorithms list the proper steps in the proper order and are detailed enough to accomplish the task.

Self-Check

1-4 Cake recipes typically omit a very important activity. Describe an activity that is missing from the algorithm above.

An algorithm often contains a step without much detail. For example, step 3, "Mix the ingredients," isn't very specific. What are the ingredients? If the problem is to write a recipe algorithm that humans can understand, step 3 should be refined a bit to instruct the cook on how to mix the ingredients. The refinement to step 3 could be something like this:

3. Empty the cake mix into the bowl and mix in the milk until smooth.

or for scratch bakers:

3a. Sift the dry ingredients.
3b. Place the liquid ingredients in the bowl.
3c. Add the dry ingredients a quarter-cup at a time, whipping until smooth.

Algorithms may be expressed in pseudocode—instructions expressed in a language that even nonprogrammers could understand. Pseudocode is written for humans, not for computers. Pseudocode algorithms are an aid to program design.

Pseudocode is very expressive. One pseudocode instruction may represent many computer instructions. Pseudocode algorithms are not concerned about issues such as misplaced punctuation marks or the details of a particular computer system. Pseudocode solutions make design easier by allowing details to be deferred. Writing an algorithm can be viewed as planning. A program developer can design with pencil and paper and sometimes in her or his head.

Algorithmic Patterns

Computer programs often require input from the user in order to compute and display the desired information. This particular flow of three activities—input/process/output—occurs so often, in fact, that it can be viewed as a pattern. It is one of several algorithmic patterns acknowledged in this textbook. These patterns will help you design programs.

A pattern is anything shaped or designed to serve as a model or a guide in making something else [Funk/Wagnalls 1968]. An algorithmic pattern serves as a guide to help develop programs. For instance, the following Input/Process/Output (IPO) pattern can be used to help design your first programs. In fact, this pattern will provide a guideline for many programs.

Algorithmic Pattern: Input Process Output (IPO)

Pattern: Input/Process/Output (IPO)

Problem: The program requires input from the user in order to compute and display the desired information.

Outline:
1. Obtain the input data.
2. Process the data in some meaningful way.
3. Output the results.

This algorithmic pattern is the first of several. In subsequent chapters, you'll see other algorithmic patterns, such as Guarded Action and Indeterminate Loop. To use an algorithmic pattern effectively, you should first become familiar with it. Look for the Input/Process/Output algorithmic pattern while developing programs. This could allow you to design your first programs more easily. For example, if you discover you have no meaningful values for the input data, it may be because you have placed the process step *before* the input step. Alternately, you may have skipped the input step altogether.

Consider this quote from Christopher Alexander's book A Pattern Language:

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Alexander is describing patterns in the design of furniture, gardens, buildings, and towns, but his description of a pattern can also be applied to program development. The IPO pattern frequently pops up during program design.

An Example of Algorithm Design

The Input/Process/Output pattern guides the design of the algorithm that relates to our `courseGrade` problem.

Three-Step Pattern	Pattern Applied to a Specific Algorithm
1. Input	1. Read in `test1`, `test2`, and `finalExam`
2. Process	2. Compute `courseGrade`
3. Output	3. Display `courseGrade`

Although algorithm development is usually an iterative process, a pattern helps to quickly provide an outline of the activities necessary to solve the `courseGrade` problem.

Self-Check

1-5 Read the three activities of the algorithm above. Do you detect a missing activity?

1-6 Read the three activities of the algorithm above. Do you detect any activity out of order?

1-7 Would this previous algorithm work if the first two activities were switched?

1-8 Is there enough detail in this algorithm to correctly compute `courseGrade`?

There currently is not enough detail in the process step of the `courseGrade` problem. The algorithm needs further refinement. Specifically, exactly how should the input data be processed to compute the course grade? The algorithm omits the weighted scale specified in the problem specification. The process step should be refined a bit more. Currently, this pseudocode algorithm does not describe how `courseGrade` must be computed.

The refinement of this algorithm (below) shows a more detailed process step. The step "Compute `courseGrade`" is now replaced with a refinement—a more detailed and specific activity. The input and output steps have also been refined. This is the design phase result in an algorithm with enough detail to begin the next phase, implementation.

Refinement of a Specific Input/Process/Output (IPO) Algorithm

- Obtain `test1`, `test2`, and `finalExam` from the user
- Compute `courseGrade` = (25% of `test1`) + (25% of `test2`) + (50% of `finalExam`)
- Display the value of `courseGrade`

Programs can be developed more quickly and with fewer errors by reviewing algorithms before moving on to the implementation phase. Are the activities in the proper order? Are all the necessary activities present?

A **computer** is a programmable electronic device that can store, retrieve, and process data. Programmers can simulate an electronic version of the algorithm by following the algorithm and manually performing the activities of storing, retrieving, and processing data using pencil and

paper. The following algorithm walkthrough is a human (non-electronic) execution of the algorithm:

1. Retrieve some example values from the user and store them as shown:

```
test1:       80
test2:       90
finalExam:   100
```

2. Retrieve the values and compute `courseGrade` as follows:

```
courseGrade = (0.25 x test1) + (0.25 x test2) + (0.50 x finalExam)
              (0.25 x 80.0)  + (0.25 x 90.0)  + (0.50 x 100.0)
                  20.0       +     22.5       +     50.0
courseGrade = 92.5
```

3. Show the course grade to the user by retrieving the data stored in `courseGrade` to show `92.5%`.

It has been said that good artists know when to put down the brushes. Deciding when a painting is done is critical for its success. By analogy, a designer must decide when to stop designing. This is a good time to move on to the third phase of program development. In summary, here is what has been accomplished so far:

- The problem is understood.
- Data have been identified and named.
- Output for two sample problems is known (`80.25%` and now `92.5%`).
- An algorithm has been developed.
- Walking through the algorithm simulated computer activities.

Implementation *(accomplishment, fulfilling, making good, execution)*

The analysis and design of simple problems could be done with pencil and paper. The implementation phase of program development requires both software (the program) and hardware (the computer). The goal of the implementation phase is to develop a program that runs correctly on a computer. **Implementation** is the collection of activities required to complete the program so someone else can use it. Here are some implementation phase activities:

Activity	What you get
Translate an algorithm into a programming language.	Source code
Compile source code into byte code.	Byte code
Run the program.	A running program
Verify that the program does what it is supposed to do.	A grade (or a satisfied customer)

Whereas the design phase provided a solution in the form of a pseudocode algorithm, the implementation phase requires nitty-gritty details. The programming language translation must be written in a precise manner according to the syntax rules of that programming language. Attention must be paid to the placement of semicolons, commas, and periods. For example, an algorithmic statement such as this:

Display the value of `courseGrade`

could be translated into Java source code that might look like this:

```
System.out.println("Course Grade: " + courseGrade + "%");
```

This output step generates output to the computer screen that might look like this (assuming the state of

`courseGrade` is 92.5):

```
Course Grade: 92.5%
```

Once a programmer has translated the user's needs into pseudocode and then into a programming language, software is utilized to translate your instructions into the lower levels of the computer. Fortunately, there is a tool for performing these translations. Programmers use a compiler to translate the high-level programming language source code (such as Java) into its byte code equivalent. This byte code can then be sent to any machine with a Java virtual machine (JVM). The Java virtual machine then converts the byte code into the machine language of that particular machine. In this way, the same Java program can run on a variety of platforms such as Unix, Mac OS, Linux, and Windows. Finally, to verify that the program works, the behavior of the executable program must be observed. Input data may be entered, and the corresponding output is observed. The output is compared to what was expected. If the two match, the program works for at least one particular set of input data. Other sets of input data can be entered while the program is running to build confidence that the program works as defined by the problem specification. Program development is summarized as shown to the right (at least this is one opinion/summary).

Although you will likely use the same compiler as in industry, the roles of people will differ. In large software organizations, many people—usually in teams—perform analysis, design, implementation, and testing. In many of these simple textbook problems, the user needs are what your instructor requires, usually for grade assessment. You will often play the role of analyst, designer, programmer, *and* tester—perhaps as part of a team, but for the most part by yourself.

Self-Check

1-9 Review the above figure and list the phases that are **-a** primarily performed by humans and **-b** primarily performed by software. Select your answers from the set of I, II, III, IV, V, and VI.

A Preview of a Java Implementation

The following program—a complete Java translation of the algorithm—previews many programming language details. You are not expected to understand this Java code. The details are presented in Chapter 2. For now, just peruse the Java code as an implementation of the pseudocode algorithm. The three variables `test1`, `test2`, and `finalExam` represent user input. The output variable is named `courseGrade`. User input is made possible through a `Scanner` (discussed in Chapter 2).

```java
// This program computes and displays a final course grade as a
// weighted average after the user enters the appropriate input.
import java.util.Scanner;

public class TestCourseGrade {

  public static void main(String[] args) {
    System.out.println("This program computes a course grade when");
    System.out.println("you have entered three requested values.");

    // I)nput test1, test2, and finalExam.
    Scanner keyboard = new Scanner(System.in);

    System.out.print("Enter first test: ");
    double test1 = keyboard.nextDouble();
    System.out.print("Enter second test: ");
    double test2 = keyboard.nextDouble();
```

```
    System.out.print("Enter final exam: ");
    double finalExam = keyboard.nextDouble();

    // P)rocess
    double courseGrade = (0.25 * test1) + (0.25 * test2) + (0.50 * finalExam);

    // O)utput the results
    System.out.println("Course Grade: " + courseGrade + "%");
  }
}
```

Dialogue

```
This program computes a course grade when
you have entered three requested values.
Enter first test: 80.0
Enter second test: 90.0
Enter final exam: 100.0
Course Grade: 92.5%
```

Testing

Although this "Testing" section appears at the end of our first example of program development, don't presume that testing is deferred until implementation. The important process of **testing** may, can, and should occur at any phase of program development. The actual work can be minimal, and it's worth the effort. However, you may not agree until you have experienced the problems incurred by *not* testing.

Testing During All Phases of Program Development

- During analysis, establish sample problems to confirm your understanding of the problem.
- During design, walk through the algorithm to ensure that it has the proper steps in the proper order.
- During testing, run the program (or method) several times with different sets of input data. Confirm that the results are correct.
- Review the problem specification. Does the running program do what was requested?
- In a short time you will see how a newer form of unit testing will help you develop software.

You should have a sample problem before the program is coded—not after. Determine the input values and what you expect for output.

When the Java implementation finally does generate output, the predicted results can then be compared to the output of the running program. Adjustments must be made any time the predicted output does not match the program output. Such a conflict indicates that the problem example, the program output, or perhaps both are incorrect. Using problem examples helps avoid the misconception that a program is correct just because the program runs successfully and generates output. The output could be wrong! Simply executing doesn't make a program right.

Even exhaustive testing does not prove a program is correct. E. W. Dijkstra has argued that testing only reveals the presence of errors, not the absence of errors. Even with correct program output, the program is not proven correct. Testing reduces errors and increases confidence that the program works correctly.

In Chapter 3, you will be introduced to an industry level testing tool that does not require user input. You will be able to build reusable automated tests. In Chapter 2, the program examples will have user input and output that must be compared manually (not automatically).

Self-Check

1-10 If the programmer predicts `courseGrade` should be `100.0` when all three inputs are `100.0` and the program displays `courseGrade` as `75.0`, what is wrong: the prediction, the program, or both?

1-11 If the programmer predicts `courseGrade` should be `90.0` when `test1` is `80`, `test2` is `90.0`, and `finalExam` is `100.0` and the program outputs `courseGrade` as `92.5`, what is wrong: the prediction, the program, or both?

1-12 If the programmer predicts `courseGrade` should be `92.5` when `test1` is `80`, `test2` is `90.0`, and `finalExam` is `100.0` and the program outputs `courseGrade` as `90.0`, what is wrong: the prediction, the program, or both

Answers to Self-Check Questions

1-1 Input: `pounds` and perhaps `todaysConversionRate`, Output: `USDollars`

1-2 `CDCollection`, `currentSelection`

1-3

Problem	Data Name	Input or Output	Sample Problem
Compute the	presentValue	Input	1000.00
future value of	periods	Input	360 (30 years)
an investment	monthlyInterestRate	Input	0.0075 (9%/year)
	futureValue	Output	14730.58

1-4 Turn the oven off (or you might recognize some other activity or detail that was omitted).

1-5 No (at least the author thinks it's okay)

1-6 No (at least the author thinks it's okay)

1-7 No. The `courseGrade` would be computed using undefined values for `test1`, `test2`, and `finalExam`.

1-8 No. The details of the process step are not present. The formula is missing.

1-9 -a I, II, III, and VI
-b IV and V

1-10 The program is wrong.

1-11 The prediction is wrong. The problem asked for a weighted average, not a simple average.

1-12 The program is wrong.

Chapter 2

Java Fundamentals

Goals

- Introduce the Java syntax necessary to write programs
- Be able to write a program with user input and console output
- Evaluate and write arithmetic expressions
- Use a few of Java's types such as `int` and `double`

2.1 Elements of Java Programming

The essential building block of Java programs is the **class**. In essence, a Java class is a sequence of characters (text) stored as a file, whose name always ends with `.java`. Each class is comprised of several elements, such as a **class heading** (`public class` *`class-name`*) and **methods**—a collection of statements grouped together to provide a service. Below is the general form for a Java class that has one method: `main`. Any class with a `main` method, including those with only a `main` method, can be run as a program.

***General Form:* A simple Java program (only one class)**

```
// Comments: any text that follows // on the same line
import package-name.class-name;

public class class-name {

  public static void main(String[] args) {
     variable declarations and initializations
     messages and operations such as assignments
  }
}
```

General forms describe the **syntax** necessary to write code that compiles. The general forms in this textbook use the following conventions:

- Boldface elements must be written exactly as shown. This includes words such as **`public static void main`** and symbols such as **`[`**, **`]`**, **`(`**, and).
- Italicized items are defined somewhere else or must be supplied by the programmer.

A Java Class with One Method Named `main`

```
// Read a number and display that input value squared
import java.util.Scanner;

public class ReadItAndSquareIt {

  public static void main(String[] args) {
    // Allow user input from the keyboard
    Scanner keyboard = new Scanner(System.in);

    // I)nput Prompt user for a number and get it from the keyboard
    System.out.print("Enter an integer: ");
    int number = keyboard.nextInt();

    // P)rocess
    int result = number * number;

    // O)utput
    System.out.println(number + " squared = " + result);
  }
}
```

Dialog

```
Enter an integer: -12
-12 squared = 144
```

The first line in the program shown above is a comment indicating what the program will do. Comments in Java are always preceded by the // symbol, and are "ignored" by the program. The next line contains the word `import`, which allows a program to use classes stored in other files. This program above has access to a class named `Scanner` for reading user input. If you omit the import statement, you will get this error:

```
Scanner keyboard = new Scanner(System.in);
Scanner cannot be resolved to a type
```

Java classes, also known as types, are organized into over seventy packages. Each package contains a set of related classes. For example, `java.net` has classes related to networking, and `java.io` has a collection of classes for performing input and output. To use these classes, you could simply use the import statement. Otherwise you would have to precede the class name with the correct package name, like this:

```
java.util.Scanner keyboard = new java.util.Scanner(System.in);
```

The next line in the sample program is a class heading. A class is a collection of methods and variables (both discussed later) enclosed within a set of matching curly braces. You may use any valid class name after `public class`; however, the class name must match the file name. Therefore, the preceding program must be stored in a file named `ReadItAndSquareIt.java`.

The file-naming convention

class-name`.java`

The next line in the program is a method heading that, for now, is best retyped exactly as shown (an explanation–intentionally skipped here–is required to have a program):

```
public static void main(String[] args)     // Method heading
```

The opening curly brace begins the body of the `main` method, which is a collection of executable statements and variables. This `main` method body above contains a variable declaration, variable

initializations, and four messages, all of which are described later in this chapter. When run as a program, the first statement in main will be the first statement executed. The body of the method ends with a closing curly brace.

This Java source code represents input to the Java compiler. A compiler is a program that translates source code into a language that is closer to what the computer hardware understands. Along the way, the compiler generates error messages if it detects a violation of any Java syntax rules in your source code. Unless you are perfect, you will see the compiler generate errors as the program scans your source code.

Tokens — The Smallest Pieces of a Program

As the Java compiler reads the source code, it identifies individual **tokens**, which are the smallest recognizable components of a program. Tokens fall into four categories:

Token	Examples
Special symbols	`; () , . { }`
Identifiers	`main args credits courseGrade String List`
Reserved identifiers	`public static void class double int`
Literals (constant values)	`"Hello World!" 0 -2.1 'C' true`

Tokens make up more complex pieces of a program. Knowing the types of tokens in Java should help you to:

- More easily write syntactically correct code.
- Better understand how to fix syntax errors detected by the compiler.
- Understand general forms.
- Complete programs more quickly and easily.

Special Symbols

A special symbol is a sequence of one or two characters, with one or possibly many specific meanings. Some special symbols separate other tokens, for example: `{`, `;`, and `,`. Other special symbols represent operators in expressions, such as: `+`, `-`, and `/`. Here is a partial list of single-character and double-character special symbols frequently seen in Java programs:

```
()   .   +   -   /   *   <=   >=   //   { }   ==   ;
```

Identifiers

Java **identifiers** are words that represent a variety of things. `String`, for example is the name of a class for storing a string of characters. Here are some other identifiers that Java has already given meaning to:

```
sqrt  String  get  println  readLine  System  equals  Double
```

Programmers must often create their own identifiers. For example, `test1`, `finalExam`, `main`, and `courseGrade` are identifiers defined by programmers. All identifiers follow these rules.

- Identifiers begin with upper- or lowercase letters `a` through `z` (or `A` through `Z`), the dollar sign `$`, or the underscore character `_`.
- The first character may be followed by a number of upper- and lowercase letters, digits (`0` through `9`), dollar signs, and underscore characters.
- Identifiers are case sensitive; `Ident`, `ident`, and `iDENT` are three different identifiers.

Valid Identifiers

```
main      ArrayList         incomeTax          MAX_SIZE           $Money$
Maine     URL               employeeName       all_4_one          _balance
miSpel    String            A1                 world_in_motion    balance
```

Invalid Identifiers

```
1A           // Begins with a digit
miles/Hour   // The / is not allowed
first Name   // The blank space not allowed
pre-shrunk   // The operator - means subtraction
```

Java is case sensitive. For example, to run a class as a program, you must have the identifier `main`. `MAIN` or `Main` won't do. The convention employed by Java programmers is to use the "camelBack" style for variables. The first letter is always lowercase, and each subsequent new word begins with an uppercase letter. For example, you will see `letterGrade` rather than `lettergrade`, `LetterGrade`, or `letter_grade`. Class names use the same convention, except the first letter is also in uppercase. You will see `String` rather than `string`.

Reserved Identifiers

Reserved identifiers in Java are identifiers that have been set aside for a specific purpose. Their meanings are fixed by the standard language definition, such as `double` and `int`. They follow the same rules as regular identifiers, but they cannot be used for any other purpose. Here is a partial list of Java reserved identifiers, which are also known as keywords.

Java Keywords

```
boolean      default      for          new
break        do           if           private
case         double       import       public
catch        else         instanceof   return
char         extends      int          void
class        float        long         while
```

The case sensitivity of Java applies to keywords. For example, there is a difference between `double` (a keyword) and `Double` (an identifier, not a keyword). All Java keywords are written in lowercase letters.

Literals

A literal value such as 123 or -94.02 is one that cannot be changed. Java recognizes these numeric literals and several others, including `String` literals that have zero or more characters enclosed within a pair of double quotation marks.

```
"Double quotes are used to delimit String literals."
"Hello, World!"
```

Integer literals are written as numbers without decimal points. Floating-point literals are written as numbers with decimal points (or in exponential notation: $5e3 = 5 * 10^3 = 5000.0$ and $1.23e-4 = 1.23 \times 10^{-4} = 0.0001234$). Here are a few examples of integer, floating-point, string, and character literals in Java, along with both Boolean literals (`true` and `false`) and the `null` literal value.

The Six Types of Java Literals

Integer	Floating Point	String	Character	Boolean	Null
-2147483648	-1.0	"A"	'a'	true	null
-1	0.0	"Hello World"	'0'	false	
0	39.95	"\n new line"	'?'		
1	1.23e09	"1.23"	' '		
2147483647	-1e6	"The answer is: "	'7'		

Note: Other literals are possible such as `12345678901L` for integers > 2,147,483,647.

Comments

Comments are portions of text that annotate a program, and fulfill any or all of the following expectations:

- Provide internal documentation to help one programmer read and understand another's program.
- Explain the purpose of a method.
- Describe what a method expects of the input arguments (n must be > 0, for example).
- Describe a wide variety of program elements.

Comments may be added anywhere within a program. They may begin with the two-character special symbol `/*` when closed with the corresponding symbol `*/`.

```
/*
   A comment may extend over many lines
   when using slash start at the beginning
   and ending the comment with a star slash.
*/
```

An alternate form for comments is to use `//` before a line of text. Such a comment may appear at the beginning of a line, in which case the entire line is "ignored" by the program, or at the end of a line, in which case all code prior to the special symbol will be executed.

```
// This Java program displays "hello, world to the console.
```

Comments can help clarify and document the purpose of code. Using intention-revealing identifiers and writing code that is easy to understand, however, can also do this.

Self-Check

2-1 List each of the following as a valid identifier or explain why it is not valid.

-a	`abc`	-i	`H.P.`
-b	`123`	-j	`double`
-c	`ABC`	-k	`55_mph`
-d	`_.$`	-l	`sales Tax`
-e	`my Age`	-m	`$$$$`
-f	`identifier`	-n	`_____`
-g	`(identifier)`	-o	`Mile/Hour`
-h	`mispellted`	-p	`Scanner`

2-2 Which of the following are valid Java comments?

-a `// Is this a comment?`

-b `/ / Is this a comment?`

-c `/* Is this a comment?`

-d `/* Is this a comment? */`

2.2 Java Types

Java has two types of variables: primitive types and reference types. Reference variables store information necessary to locate complex values such as strings and arrays. On the other hand, Primitive variables store a single value in a fixed amount of computer memory. The eight "primitive" (simple) types are closely related to computer hardware. For example, an `int` value is stored in 32 bits (4 bytes) of memory. Those 32 bits represent a simple positive or negative integer value. Here is summary of all types in Java along with the range of values for the primitive types:

The Java Primitive Types

integers:
byte (8 bits) -128 .. 128
short (16 bits) -32,768 .. 32,767
int (32 bits) -2,147,483,648 .. 2,147,483,647
long (64 bits) -9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807

real numbers:
float (32 bits), ±1.40129846432481707e-45 .. ±3.40282346638528860e+38
double (64 bits), ±4.94065645841246544e-324 .. ±1.79769313486231570e+308

other primitive types:
char 'A', '@', or 'z' for example
boolean has only two literal values **false** and **true**

The Java Reference Types

classes
arrays
interfaces

Declaring a primitive variable provides the program with a named data value that can change while the program is running. An initialization allows the programmer to set the original value of a variable. This value can be accessed or changed later in the program by using the variable name.

General Form: **Initializing (declaring a primitive variable and giving it a value)**

type identifier; `// Declare one variable`

type identifier = *initial-value*; `// For primitive types like int and double`

Example: The following code declares one `int` and two `double` primitive variables while it initializes `grade`.

```
int credits;
double grade = 4.0;
double GPA;
```

The following table summarizes the initial value of these variables:

Variable Name	Value
`credits`	`?  // Unknown`
`grade`	`4.0  // This was initialized above`
`GPA`	`?  // Unknown`

If you do not initialize a variable, it cannot be used unless it is changed with an assignment statement. The Java compiler would report this as an error.

Assignment

An **assignment** gives a value to a variable. The value of the expression to the right of the assignment operator (=) replaces the value of the variable to the left of =.

***General Form:* Assignment**

variable-name **=** *expression* **;**

The *expression* must be a value that can be stored by the type of variable to the left of the assignment operator (=). For example, an expression that results in a floating-point value can be stored in a `double` variable, and likewise an integer value can be stored in an `int` variable.

```
int credits = 4;
double grade = 3.0;
double GPA = (credits * grade) / credits; // * and / evaluate before =
```

The assignment operator = has a very low priority, it assigns after all other operators evaluate. For example, `(credits * grade) / credits` evaluates to 3.0 before 3.0 is assigned to `GPA`. These three assignments change the value of all three variables. The values can now be shown like this:

Variable	Value
`credits`	`4`
`grade`	`3.0`
`GPA`	`3.0`

In an assignment, the Java compiler will check to make sure you are assigning the correct type of value to the variable. For example, a string literal cannot be assigned to a numeric variable. A floating-point number cannot be stored in an `int`.

```
grade = "Noooooo, you can't do that"; // Cannot store string in a double
credits = 16.5; // Cannot store a floating-point number in an int
```

Self-Check

2-3 Which of the following are valid attempts at assignment, given these two declarations?

```
double aDouble = 0.0;
int anInt = 0;
```

-a	`anInt = 1;`	**-e**	`aDouble = 1;`
-b	`anInt = 1.5;`	**-f**	`aDouble = 1.5;`
-c	`anInt = "1.5";`	**-g**	`aDouble = "1.5";`
-d	`anInt = anInt + 1;`	**-h**	`aDouble = aDouble + 1.5;`

Input and Output (I/O)

Programs communicate with users. Such communication is provided through—but is not limited to—keyboard **input** and screen **output**. In Java, this two-way communication is made possible by sending messages, which provide a way to transfer control to another method that performs some well-defined responsibility. You may have written that method, or it may very likely be a method you cannot see in one of the existing Java classes. Some messages perform particular actions. Two such methods are the `print` and `println` messages sent to `System.out`:

***General Form:* Output with print and println**

`System.out.print(`*expression***`);`**
`System.out.println(`*expression***`);`**

`System.out` is an existing reference variable that represents the console—the place on the computer screen where text is displayed (not actually printed). The expression between the parentheses is known as the **argument**. In a print or println message, the value of the expression will be displayed on the computer screen. With `print` and `println`, the arguments can be any of the types mentioned so far (`int`, `double`, `char`, `boolean`), plus others. The semicolon (`;`) terminates messages. The only difference between `print` and `println` is that `println` generates a new line. Subsequent output begins at the beginning of a new line. Here are some valid output messages:

```
System.out.print("Enter credits: ");  // Use print to prompt the user
System.out.println();                 // Print a blank line
```

Input

To make programs more applicable to general groups of data—for example, to compute the GPA for any student—variables are often assigned values through keyboard input. This allows the program to accept data which is specific to the user. There are several options for obtaining user input from the keyboard. Perhaps the simplest option is to use the `Scanner` class from the `java.util` package. This class has methods that allow for easy input of numbers and other types of data, such as strings.

Before you can use `Scanner` messages such as `nextDouble` or `nextInt`, your code must create a reference variable to which messages can be sent. The following code initializes a reference variable named `keyboard` that will allow the keyboard to be a source of input. (`System.in` is an existing reference variable that allows characters to be read from the keyboard.)

Creating an Instance of `Scanner` to Read Numeric Input

```
// Store a reference variable named keyboard to read input from the user.
// System.in is a reference variable already associated with the keyboard
Scanner keyboard = new Scanner(System.in);
```

In general, a reference variable is initialized with the keyword `new` followed by *class-name* and *(initial-values)*.

***General Form:* Initializing reference variables with `new`**

class-name reference-variable-name **`= new`** *class-name* **`();`**
class-name reference-variable-name **`= new`** *class-name* **`(`***initial-value(s)***`);`**

The expression to the right of = evaluates to a reference value, which is then stored in the reference variable to the left of `=`. That reference value is used later for sending messages. Messages sent to `keyboard` can obtain textual input from the keyboard and can convert that text (for example, 3.45 and 99) into numbers. Here are two messages that allow users to input numbers into a program:

Numeric Input

```
keyboard.nextInt();     // Pause until user enters an integer
keyboard.nextDouble(); // Pause until user enters a floating-point number
```

In general, use this form to send a message to a reference variable that will, in turn, cause some operation to execute:

***General Form:* Sending messages**

reference-variable-name **`.`***message-name* **`(`***argument-list***`)`**

When a `nextInt` or `nextDouble` message is sent to `keyboard`, the method waits until the user enters some type of input and then presses the Enter key. If the user enters the number correctly, the text will be converted into the proper machine representation of the number. If the user enters a letter when `keyboard` is expecting a number, the program may terminate with an error message.

These two methods are examples of expressions that evaluate to some value. Whereas a `nextInt` message evaluates to a primitive `int` value, a `nextDouble` message evaluates to a primitive floating-point value. Because `nextInt` and `nextDouble` return numeric values, they are often seen on the right-hand side of assignment statements. These messages will be seen in text-based input and output programs (ones that have no graphical user interface).

For example, the following code prompts the user to enter two numbers using `print`, `nextInt`, and `nextDouble` messages.

```
System.out.print("Enter credits: ");     // Prompt the user
credits = keyboard.nextInt();            // Read and assign an integer
System.out.print("Enter grade: ");       // Prompt the user
qualityPoints = keyboard.nextDouble();   // Read and assign a double
```

Dialog

```
Enter credits: 4
Enter grade: 3.0
```

In the last line of code above—the fourth message—the `nextDouble` message causes a pause in program execution until the user enters a number. When the user types a number and presses the enter key, the `nextDouble` method converts the text user into a floating-point number. That value is then assigned to the variable `qualityPoints`. All of this happens in one line of code.

Prompt then Input

The output and input operations are often used together to obtain values from the user of the program. The program informs the user what must be entered with an output message and then sends an input message to get values for the variables. This happens so often that this activity can be considered to be a pattern. The **Prompt then Input** pattern has two activities:

1. Request the user to enter a value (prompt).
2. Obtain the value for the variable (input).

Algorithmic Pattern: Prompt and Input

Pattern:	Prompt and Input
Problem:	The user must enter something.
Outline:	1. Prompt the user for input. 2. Input the data
Code Example:	`System.out.println("Enter credits: ");` `int credits = keyboard.nextInt();`

Strange things may happen if the prompt is left out. The user will not know what must be entered. Whenever you require user input, make sure you prompt for it first. Write the code that tells the user precisely what you want. First output the prompt and then obtain the user input. Here is another instance of the Prompt and Input pattern:

```
System.out.println("Enter test #1: ");
double test1 = keyboard.nextDouble(); // Initialize test1 with input
System.out.println("You entered " + test1);
```

Dialogue

```
Enter test #1: 97.5
You entered 97.5
```

In general, tell the user what value is needed, then input a value into that variable with an input message such as `keyboard.nextDouble();`.

***General Form:* Prompt and Input**

`System.out.println("`*prompt user for input :*** **`" );`**
input **`= keyboard.nextDouble();`** `// or keyboard.nextInt();`

Arithmetic Expressions

Arithmetic expressions are made up of two components: operators and operands. An arithmetic operator is one of the Java special symbols `+`, `-`, `/`, or `*`. The operands of an arithmetic expression may be numeric variable names, such as `credits`, and numeric literals, such as `5` and `0.25`.

An Arithmetic Expression may be	Example
numeric variable	`double aDouble`
numeric literal	`100` or `99.5`
expression + expression	`aDouble + 100`
expression – expression	`aDouble - 100`
expression * expression	`aDouble * 100`
expression / expression	`aDouble / 99.5`
(expression)	`(aDouble + 2.0)`

The last definition of "expression" suggests that we can write more complex expressions.

```
1.5 * ((aDouble - 99.5) * 1.0 / aDouble)
```

Since arithmetic expressions may be written with many literals, numeric variable names, and operators, rules are put into force to allow a consistent evaluation of expressions. The following table lists four Java arithmetic operators and the order in which they are applied to numeric variables.

Most Arithmetic Operators

`* / %`	In the absence of parentheses, multiplication and division evaluate before addition and subtraction. In other words, *, /, and % have precedence over + and -. If more than one of these operators appears in an expression, the leftmost operator evaluates first.
`+ -`	In the absence of parentheses, + and - evaluate after all of the *, /, and % operators, with the leftmost evaluating first. Parentheses may override these precedence rules.

The operators of the following expression are applied to operands in this order: `/`, `+`, `-`.

```
2.0 + 5.0 - 8.0 / 4.0    // Evaluates to 5.0
```

Parentheses may alter the order in which arithmetic operators are applied to their operands.

```
(2.0 + 5.0 - 8.0) / 4.0      // Evaluates to -0.25
```

With parentheses, the `/` operator evaluates last, rather than first. The same set of oper-ators and operands, with parentheses added, has a different result (`-0.25` rather than `5.0`).

These precedence rules apply to binary operators only. A binary operator is one that requires one operand to the left and one operand to the right. A unary operator requires one operand on the right. Consider this expression, which has the binary multiplication operator `*` and the unary minus operator `-`.

```
3.5 * -2.0  // Evaluates to -7.0
```

The unary operator evaluates before the binary * operator: 3.5 times negative 2.0 results in negative 7.0. Two examples of arithmetic expressions are shown in the following complete program that computes the GPA for two courses.

```
// This program calculates the grade point average (GPA) for two courses.
import java.util.Scanner;

public class TwoCourseGPA {

  public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);

    // Prompt and Input the credits and grades for two courses
    System.out.print("Enter credits for first course: ");
    double credits1 = keyboard.nextDouble();
    System.out.print("Enter grade for first course: ");
    double grade1 = keyboard.nextDouble ();
    System.out.print("Enter credits for second course: ");
    double credits2 = keyboard.nextDouble ();
    System.out.print("Enter grade for second course: ");
    double grade2 = keyboard.nextDouble();

    // Compute the GPA
    double qualityPoints = (credits1 * grade1) + (credits2 * grade2);
    double GPA = qualityPoints / (credits1 + credits2);
```

```
    // Show the result
    System.out.println();
    System.out.print("GPA for these two courses: ");
    System.out.println(GPA);
  }
}
```

Output

```
Enter credits for first course: 3.0
Enter grade for first course: 4.0
Enter credits for second course: 2.0
Enter grade for second course: 3.0

GPA for these two courses: 3.6
```

Self-Check

2-4. Write a complete Java program that prompts for a number from 0.0 to 1.0 and echos (prints) the user's input. The dialog generated by your program should look like this:

```
Enter relativeError [0.0 through 1.0]: 0.341
You entered: 0.341
```

2-5. Write the output generated by the following program:

```
public class Arithmetic {
  public static void main(String[] args) {
    double x = 1.2;
    double y = 3.4;
    System.out.println(x + y);
    System.out.println(x - y);
    System.out.println(x * y);
  }
}
```

2-6. Write the complete dialog (program output and user input) generated by the following program when the user enters each of these input values for `sale`:

a. ***10.00*** b. ***12.34*** c. ***100.00***

```
import java.util.Scanner;

public class InputProcessOutput  {

  public static void main(String[] args)  {
    double sale = 0.0;
    double tax = 0.0;
    double total = 0.0;
    double TAX_RATE = 0.07;
    Scanner keyboard = new Scanner(System.in);
    // I)nput
    System.out.print("Enter sale: ");
    sale = keyboard.nextDouble();  // User enters 10.00, 12.34, or 100.00
    // P)rocess
    tax = sale * TAX_RATE;
    total = sale + tax;

    // O)utput
    System.out.println("Sale: " + sale);
    System.out.println("Tax: " + tax);
    System.out.println("Total: " + total);
  }
}
```

2-7 Evaluate the following arithmetic expressions:

```
double x = 2.5;
double y = 3.0;
```

-a	`x * y + 3.0`	**-d**	`1.5 * ( x - y )`
-b	`0.5 + x / 2.0`	**-e**	`y + -x`
-c	`1.0 + x * 3.0 / y`	**-f**	`( x - 2.0 ) * ( y - 1.0 )`

`int` Arithmetic

A variable declared as `int` can store a limited range of whole numbers (numbers without fractions). Java `int` variables store integers in the range of -2,147,483,648 through 2,147,483,647 inclusive. All `int` variables have operations similar to `double` (+, *, -, =), but some differences do exist, and there are times when `int` is the correct choice over `double`. For example, a fractional remainder cannot be stored in an `int`. In fact, you cannot assign a floating-point literal or `double` variable to an `int` variable. If you do, the compiler complains with an error.

```
int anInt = 1.999;        // ERROR
int anotherInt = 0.0;     // ERROR
```

The / operator has different meanings for `int` and `double` operands. Whereas the result of 3 / 4 is 0, the result of 3.0 / 4.0 is 0.75. Two integer operands with the / operator have an integer result—not a floating-point result, as in the latter example. When writing programs, remember to choose an `int` or `double` data type correctly, in order to appropriately reflect the type of value you would like to store.

The remainder operation—symbolized with the % `(modulus)` operator—is also available for both `int` and `double operands`. For example, the result of 18 % 4 is the integer remainder after dividing 18 by 4, which is 2. Integer arithmetic is illustrated in the following code, which shows % and / operating on integer expressions, and / operating on floating-point operands. In this example, the integer results describe whole hours and whole minutes rather than the fractional equivalent.

```
// Show quotient remainder division with / and %
public class DivMod  {
  public static void main(String[] args)  {
    int totalMinutes = 254;
    int hours = totalMinutes / 60;
    int minutes = totalMinutes % 60;
    System.out.println(totalMinutes + " minutes can be rewritten as ");
    System.out.println(hours + " hours and " + minutes + " minutes");
  }
}
```

Output

```
254 minutes can be rewritten as
4 hours and 14 minutes
```

The preceding program indicates that even though `int`s and `double`s are similar, there are times when `double` is the more appropriate type than `int`, and vice versa. The `double` type should be specified when you need a numeric variable with a fractional component. If you need a whole number, select `int`.

Mixing Integer and Floating-Point Operands

Whenever integer and floating-point values are on opposite sides of an arithmetic operator, the integer operand is **promoted** to its floating-point equivalent. The integer 6, for example, becomes 6.0, in the case of 6 / 3.0. The resulting expression is then a floating-point number, 2.0. The same rule applies when one operand is an `int` variable and the other a `double` variable. Here are a few examples of expression with the operands are a mix of `int` and `double`.

```
public class MixedOperands {
  public static void main(String[] args)  {
    int number = 9;
    double sum = 567.9;
    System.out.println(sum / number);  // Divide a double by an int
    System.out.println(number / 2);    // Divide an int by an int
    System.out.println(number / 2.0);  // Divide an int by a double
    System.out.println(2.0 * number);  // Result is a double: 18.0 not 18
  }
}
```

Output

```
63.099999999999994
4
4.5
18.0
```

Expressions with more than two operands will also evaluate to floating-point values if one of the operands is floating-point—for example, (8.8/4+3) = (2.2 + 3) = 5.2. Operator precedence rules also come into play—for example, (3 / 4 + 8.8) = (0 + 8.8) = 8.8.

Self-Check

2-8 Evaluate the following expressions.

-a	`5 / 9`	**-f**	`5 / 2`
-b	`5.0 / 9`	**-g**	`7 / 2.5 * 3 / 4`
-c	`5 / 9.0`	**-h**	`1 / 2.0 * 3`
-d	2 + 4 * 6 / 3	**-i**	5 / 9 * (50.0 - 32.0)
-e	(2 + 4) * 6 / 3	**-j**	5 / 9.0 * (50 - 32)

The **boolean** Type

Java has a primitive `boolean` data type to store one of two `boolean` literals: `true` and `false`. Whereas arithmetic expressions evaluate to a number, `boolean` expressions, such as `credits > 60.0`, evaluate to one of these `boolean` values. A `boolean` expression often contains one of these relational operators:

Operator	Meaning
`<`	Less than
`>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to
`==`	Equal to
`!=`	Not equal to

When a relational operator is applied to two operands that can be compared to one another, the result is one of two possible values: true or false. The next table shows some examples of simple `boolean` expressions and their resulting values.

Boolean Expression	Result
`double x = 4.0;`	
`x < 5.0`	`true`
`x > 5.0`	`false`
`x <= 5.0`	`true`
`5.0 == x`	`false`
`x != 5.0`	`true`

Like primitive numeric variables, `boolean` variables can be declared, initialized, and assigned a value. The assigned expression must be a `boolean` expression—thus, the result of the expression must also evaluate to true or false. This is shown in the initializations, assignments, and output of three boolean variables in the following code:

```
// Initialize three boolean variables to false
boolean ready = false;
boolean willing = false;
boolean able = false;
double credits = 28.5;
double hours = 9.5;
// Assign true or false to all three boolean variables
ready = hours >= 8.0;
willing = credits > 20.0;
able = credits <= 32.0;
System.out.println("ready: " + ready);
System.out.println("willing: " + willing);
System.out.println("able: " + able);
```

Output

```
ready: true
willing: true
able: true
```

Self-Check

2-9 Evaluate the following expressions to their correct value.

```
int j = 4;
int k = 7;
```

a. `(j + 4) == k`
b. `j == 0`
c. `j >= k`
d. `j != k`
e. `j < k`
f. `j == 4`
g. `j == (j + k - j)`
h. `(k - 5) <= (j + 2)`

Boolean Operators

Java has three Boolean operators `!` to represent logical not, `||` to represent logical or, and `&&` to represent logical and. These three Boolean operators allow us to write more complex boolean expressions to express our intentions. For example, this boolean expression shows the boolean "and" operator (&&) applied to two boolean operands to determine if test is in the range of 0 through 100 inclusive.

```
(test >= 0) && (test <= 100)
```

Used in assertions, this `Boolean` expression evaluates to true when test is 97 and false when test is 977:

When `test` is `97`	When `test` is `977`
`(test >= 0) && (test <= 100)`	`(test >= 0) && (test <= 100)`
`( 97  >= 0) && ( 97  <= 100)`	`(977  >= 0) && (977  <= 100)`
`true  &&  true`	`true  &&  false`
`true`	`false`

Since there are only two Boolean values, true and false, the following table shows every possible combination of Boolean values and operators `!`, `||`, and `&&`:

`!` boolean "not" operator

Expression	Result
`! false`	`true`
`! true`	`false`

`||` boolean "or" operator

Expression	Result
`true || true`	`true`
`true || false`	`true`
`false || true`	`true`
`false || false`	`false`

`&&` boolean "and" operator

Expression	Result
`true && true`	`true`
`true && false`	`false`
`false && true`	`false`
`false && false`	`false`

Precedence Rules

Programming languages have **operator precedence** rules governing the order in which operators are applied to operands. For example, in the absence of parentheses, the relational operators `>=` and `<=` are evaluated before the `&&` operator. Most operators are grouped (evaluated) in a left-to-right order: `a/b/c/d` is equivalent to `(((a/b)/c)/d)`.

Table 6.1 lists some (though not all) of the Java operators in order of precedence. The dot `.` and `()` operators are evaluated first (have the highest precedence), and the assignment operator `=` is evaluated last. This table shows all of the operators used in this textbook (however, there are more).

Precedence rules for operators some levels of priorities are not shown

Precedence	Operator	Description	Associativity
1	`.`	Member reference	Left to right
	`()`	Method call	
2	`!`	Unary logical complement ("not")	Right to left
	`+`	Unary plus	
	`-`	Unary minus	
3	`new`	Constructor of objects	
4	`*`	Multiplication	Left to right
	`/`	Division	

	`%`	Remainder	
5	`+`	Addition (for `int` and `double`)	Left to right
	`+`	String concatenation	
	`-`	Subtraction	
7	`<`	Less than	Left to right
	`<=`	Less than or equal to	
	`>`	Greater than	
	`>=`	Greater than or equal to	
8	`==`	Equal	Left to right
	`!=`	Not equal	
12	`&&`	Boolean "and"	Left to right
13	`¦¦`	Boolean "or"	Left to right
15	`=`	Assignment	Right to left

These elaborate precedence rules are difficult to remember. If you are unsure, use parentheses to clarify these precedence rules. Using parentheses makes the code more readable and therefore more understandable that is more easily debugged and maintained.

Self-Check

2-10 Evaluate the following expressions to `true` or `false`:

a. `false ¦¦ true`
b. `true && false`
c. `(1 * 3 == 4 ¦¦ 2 != 2)`
d. `false ¦¦ true && false`
e `3 < 4 && 3 != 4`
f. `!false && !true`
g. `(5 + 2 > 3 * 4) && (11 < 12)`
h. `! ((false && true) ¦¦ false)`

2-11 Write an expression that is true only when an `int` variable named `score` is in the range of 1 through 10 inclusive.

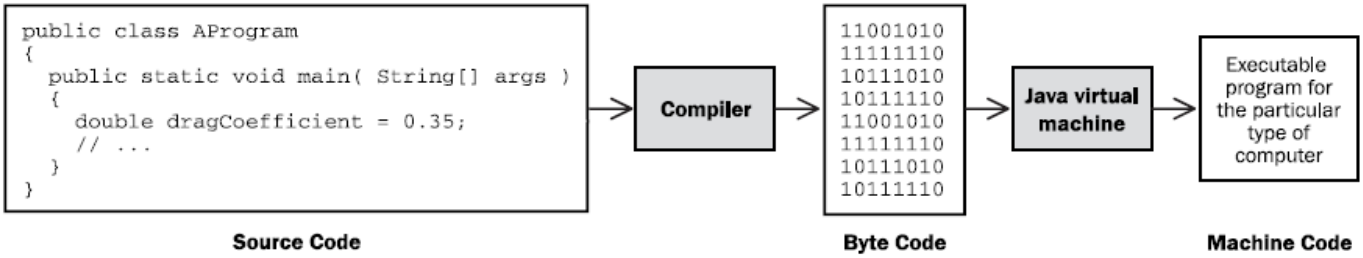
Errors

There are several categories of errors encountered when programming:

- **syntax errors**—errors that occur when compiling source code into byte code
- **intent errors**—the program does what you typed, not what you intended
- **exception**—errors that occur as the program executes

When programming, you will be writing source code using the syntax for the Java programming language. This source code is translated into byte code by the compiler, and is then stored in a `.class` file. The byte code is the same for each computer system.
For this byte code to execute, another program, called the Java virtual machine (JVM), translates the Java byte code into instructions understood by that computer. This extra step is necessary for one of the main advantages of Java: the same program can run in any computing environment!
A computer might be running Windows, MacOS, Solaris, Unix, or Linux—each computer system has its own Java virtual machine program. Having a particular Java virtual machine for each computer system also allows the same Java `.class` file to be transported around the Internet. The following figure shows the levels of translation needed in order to get executable programs to run on most computers.

From Source Code to a Program that Runs on Virtually any Computer (thanks to byte code)



1. The programmer translates algorithms into Java source code.
2. The compiler translates the source code into byte code.
3. The Java virtual machine translates byte code into the instructions understood by the computer system (Solaris, Unix, Linux, Mac OS, or Windows).

Syntax Errors Detected at Compile Time

When you are compiling source code or running your program on a computer, errors may crop up. The easiest errors to detect and fix are the errors generated by the **compiler**. These are **syntax errors** that occur during **compile time**, the time at which the compiler is examining your source code to detect and report errors, and/or to attempt to generate executable byte code from error-free source code.

A programming language requires strict adherence to its own set of formal syntax rules. It is not difficult for programmers to violate these syntax rules! All it takes is one missing `{` or `;` to foul things up. As you are writing your source code, you will often use the compiler to check the syntax of the code you wrote. While the Java compiler is translating source code into byte code so that it can run on a computer, it is also locating and reporting as many errors as possible. If you have any syntax errors, the byte code will not be generated—the program simply cannot run. If you are using the Eclipse integrated development, you will see compile time errors as you type, sometimes because you haven't finished what you were doing. To get a properly running program, you need to first correct ALL of your syntax errors.

Compilers generate many error messages. However, it is your source code that is the origin of these errors. Small typographical (and human) mistakes can be responsible for much larger roadblocks, from the compiler's perspective. Whenever your compiler appears to be nagging you, remember that the compiler is there to help you correct your errors!

The following program attempts to show several errors that the compiler should detect and report. Because error messages generated by compilers vary among systems, the reasons for the errors below are indexed with numbers to explanations that follow. Your system will certainly generate quite different error messages.

```
// This program attempts to convert pounds to UK notation.
// Several compile time errors have been intentionally retained.
public class CompileTimeErrors {

  public static void main(String[] args)  {
    System.out.println("Enter weight in pounds: ") ❶
    int pounds = keyboard.nextInt()❷;
    System.out.print("In the U.K. you weigh❸);
    System.out.print(❹Pounds / 14 + " stone, "❺pounds % 14);
  }
}
```

❶ A semicolon (`;`) is missing

❷ `keyboard` was not declared

❸ A double quote (") is missing
❹ `pounds` was written as `Pounds`
❺ The extra expressions require a missing concatenation symbol (+)

Syntax errors take some time to get used to, so try to be patient and observe the location where the syntax error occurred. The error is usually near the line where the error was detected, although you may have to fix preceding lines. Always remember to fix the first error first. An error that was reported on line 10 might be the result of a semicolon that was forgotten on line 5. The corrected source code, without error, is given next, followed by an interactive dialog (user input and computer output):

```
// This program converts pounds to the UK weight measurement.
import java.util.Scanner;

public class ErrorFree {

  public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);

    System.out.print("Enter weight in pounds: ");
    int pounds = keyboard.nextInt();
    System.out.print("In the U.K. you weigh ");
    System.out.println((pounds / 14) + " stone, " + (pounds % 14));
  }
}
```

Dialog

```
Enter weight in pounds: 146
In the U.K. you weigh 10 stone, 6
```

A different type of error occurs when `String[] args` is omitted from the `main` method:

```
  public static void main()
```

When the program tries to run, it looks for a method named `main` with `(String[] identifier)`. If you forget to write `String[] args`, you would get the error below shown after the program begins. The same error occurs if main has an uppercase M.

```
  Exception in thread "main" java.lang.NoSuchMethodError: main
```

This type of error, which occurs while the program is running, is known as an **exception**.

Exceptions

After your program compiles with no syntax errors, you will get a .class file containing the byte code that can be run on the Java virtual machine. The virtual machine can be invoked by issuing a Java command with the .class file name. For example, entering the command `java ErrorFree` at your operating system prompt will run the above program, assuming that you have a Java runtime environment (jre) installed on your computer and that the file ErrorFree.class exists.

However, when a program runs, errors may still occur. If the user enters a string that is supposed to be a number, what is the program to do? If the user enters "1oo" instead of "100" for example, is the program supposed to assume that the user meant 100? What should happen when the user enters "Kim" instead of a number? What should happen when an arithmetic expression results in division by zero? Or when there is an attempt to read from a file on a disk, but there is no disk in the drive, or the file name is wrong? Such events that occur while the program is running are known as exceptions.

One exception was shown above. The `main` method was valid, so the code compiled. However, when the program ran, Java's runtime environment was unable to locate a `main` method with

`String[] args`. The error could not be discovered until the user ran the program, at which time Java began attempted to locate the beginning of the program. If Java cannot find a method with the following line of code, a runtime exception occurs and the program terminates prematurely.

```
  public static void main(String[] args)
```

Now consider another example of an exception that occurs while the program is running. The output for the following code indicates that Java does not allow integer division by zero. The compiler does a lot of things, but it does not check the values of variables. If, at runtime, the denominator in a division happens to be 0, an `ArithmeticException` occurs.

```
public class AnArithmeticException {

  public static void main(String[] args) {
    // Integer division by zero throws an ArithmeticException
    int numerator = 5;
    int denominator = 0;
    int quotient = numerator / denominator; // A runtime error
    System.out.println("This message will not execute.");
  }
}
```

Output

```
   Exception in thread "main" java.lang.ArithmeticException: / by zero
      at A.main(A.java:8)
```

When you encounter one of these exceptions, consider the line number (`7`) where the error occurred. The reason for the exception (`/ by zero`) and the name of the exception (`ArithmeticException`) are two other clues to help you figure out what went wrong.

Intent Errors (Logic Errors)

Even when no syntax errors are found and no runtime errors occur, the program still may not execute properly. A program may run and terminate normally, but it may not be correct. Consider the following program:

```
// This program finds the average given the sum and the size
import java.util.Scanner;

public class IntentError {

  public static void main(String[] args) {
    double sum = 0.0;
    double average = 0.0;
    int number = 0;
    Scanner keyboard = new Scanner(System.in);
    // Input:
    System.out.print("Enter sum: ");
    sum = keyboard.nextDouble();
    System.out.print("Enter number: ");
    number = keyboard.nextInt();
    // Process
    average = number / sum;
    // Output
    System.out.println("Average: " + average);
  }
}
```

Dialog

```
Enter sum: 291
Enter number: 3
Average: 0.010309278350515464
```

Such intent errors occur when the program does what was typed, not what was intended. The compiler cannot detect such intent errors. The expression `number / sum` is syntactically correct—the compiler just has no way of knowing that this programmer intended to write `sum / number` instead.

Intent errors, also known as logic errors, are the most insidious and usually the most difficult errors to correct. They also may be difficult to detect—the user, tester, or programmer may not even know they exist! Consider the program controlling the Therac 3 cancer radiation therapy machine. Patients received massive overdoses of radiation resulting in serious injuries and death, while the indicator displayed everything as normal. Another infamous intent error involved a program controlling a probe that was supposed to go to Venus. Simply because a comma was missing in the Fortran source code, an American Viking Venus probe burnt up in the sun. Both programs had compiled successfully and were running at the time of the accidents. However, they did what the programmers had written—obviously not what was intended.

Answers to Self-Check Questions

2-1
- -a VALID
- -b can't start an identifier with digit 1
- -c VALID
- -d . is a special symbol.
- -e A space is not allowed.
- -f VALID
- -g ( ) are not allowed.
- -h VALID
- -i Periods (.) are not allowed.
- -j VALID
- -k Can't start identifiers with a digit.
- -l A space is not allowed.
- -m VALID but not very clear
- -n VALID but not very clear
- -o / is not allowed.
- -p VALID (but don't use it, Java already does)

2-2 Which of the following are valid Java comments?

-a	`// Is this a comment?`	Yes
-b	`/ / Is this a comment?`	No, there is a space between the slashes
-c	`/* Is this a comment?`	No, the closing `*/` is missing
-d	/* Is this a comment? */	Yes

2-3
- a VALID
- b attempts to assign a floating-point to an `int`.
- c attempts to assign a string to an `int`
- d VALID
- e VALID
- f valid
- g attempts to assign a string to a `double`.
- h VALID

2-4
```
import java.util.Scanner;
public class RelativeError {   // Your class name may vary
  public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
    System.out.print("Enter relativeError [0.0 through 1.0]: ");
    double relativeError = keyboard.nextDouble();
    System.out.print("You entered: " + relativeError);
  }
}
```

2-5
```
4.6
-2.2
4.08
```

2-6

a. ***10.00***
```
Enter sale: 10.00
Sale: 10.0
Tax: 0.7
Total: 10.7
```

b. ***12.34***
```
Enter sale: 12.34
Sale: 12.34
Tax: 0.8638
Total: 13.2038
```

c. ***100.00***
```
Enter sale: 100.00
Sale: 100.0
Tax: 7.0
Total: 107.0
```

2.7

-a 10.5 -d -0.75
-b 1.75 -e 0.5
-c 3.5 -f 1.0

2-8

-a 0 -f 2
-b 0.55556 -g 2.1
-c 0.55556 -h 1.5
-d 10 -i 0.0 5/9 is 0, 0*18.0 is 0.0
-e 12 -j 10.0

2-9

-a false -e true
-b false -f true
-c false -g false
-d true -h true

2-10

a. true e true
b. false f. false
c. false g. false
d. false h. true

2-11 (score >= 1) && (score <= 10)

Chapter 3

Objects and JUnit

Goals

This chapter is mostly about using objects and getting comfortable with sending messages to objects. Several new types implemented as Java classes are introduced to show just a bit of Java's extensive library of classes. This small subset of classes will then be used in several places throughout this textbook. You will begin to see that programs have many different types of objects. After studying this chapter, you will be able to:

- Use existing types by constructing objects
- Be able to use existing methods by reading method headings and documentation
- Introduce assertions with JUnit
- Evaluate Boolean expressions that result in true or false.

3.1 Find the Objects

Java has two types of values: primitive values and reference values. Only two of Java's eight primitive types (`int` and `double`) and only one of Java's reference types (the Scanner class) have been shown so far. Whereas a primitive variable stores only one value, a reference variable stores a reference to an object that may have many values. Classes allow programmers to model real-world entities, which usually have more values and operations than primitives.

Although the Java programming language has only eight primitive types, Java also come with thousands of reference types (implemented as Java classes). Each new release of Java tends to add new reference types. For example, instances of the Java `String` class store collections of characters to represent names and addresses in alphabets from around the world. Other classes create windows, buttons, and input areas of a graphical user interface. Other classes represent time and calendar dates. Still other Java classes provide the capability of accessing databases over networks using a graphical user interface. Even then, these hundreds of classes do not supply everything that every programmer will ever need. There are many times when programmers discover they need their own classes to model things in their applications. Consider the following system from the domain of banking:

The Bank Teller Specification

> Implement a bank teller application to allow bank customers to access bank accounts through unique identification. A customer, with the help of the teller, may complete any of the following transactions: withdraw money, deposit money, query account balances, and see the most recent 10 transactions. The system must maintain the correct balances for all accounts. The system must be able to process one or more transactions for any number of customers.

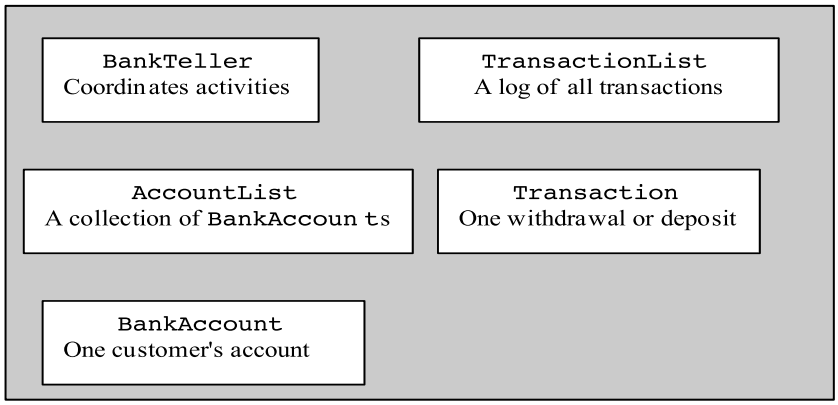
You are not asked to implement this system now. However, you should be able to pick out some things (objects) that are relevant to this system. This is the first step in the analysis phase of object-oriented software development. One simple tool for finding objects that potentially model a solution is to write down the nouns and noun phrases in the problem statement. Then consider each as a candidate object that might eventually represent part of the system. The objects used to build the system come from sources such as

- the problem statement
- an understanding of the problem domain (knowledge of the system that the problem statement may have missed or taken for granted)
- the words spoken during analysis
- the classes that come with the programming language

The objects should model the real world if possible. Here are some candidate objects:

Candidate Objects to Model a Solution	
bank teller	transaction
customers	most recent 10 transactions
bank account	window

Here is a picture to give an impression of the major objects in the bank teller system. The `BankTeller` will accomplish this by getting help from many other objects.



We now select one of these objects—`BankAccount`.

`BankAccount` Objects

Implementing a **`BankAccount`** type as a Java class gives us the ability to have many (thousands of) `BankAccount` objects. Each instance of `BankAccount` represents an account at a bank. Using your knowledge of the concept of a bank account, you might recognize that each `BankAccount` object should have its own account number and its own account balance. Other values could be part of every `BankAccount` object: a transaction list, a personal identification number (PIN), and a mother's maiden name, for example. You might visualize other banking methods, such as creating a new account, making deposits, making withdrawals, and accessing the current balance. There could also be many other banking messages—`applyInterest` and `printStatement`, for example.

As a preview to a type as a collection of methods and data, here is the `BankAccount` type implemented as a Java class and used in the code that follows. The Java class with methods and variables to implement a new type will be discussed in Chapters 4 (Methods) and 10 (Classes). Consider this class to be a blueprint that can be used to construct many `BankAccount` objects.

Each `BankAccount` object will have its their own `balance` and `ID`. Each `BankAccount` will understand the same five messages: `getID`, `equals`, `getBalance`, `deposit`, and `withdraw`.

```
// A type that models a very simple account at a bank.
public class BankAccount {

  // Values that each object "remembers":
  private String ID;
  private double balance;

  // The constructor:
  public BankAccount(String initID, double initBalance) {
    ID = initID;
    balance = initBalance;
  }

  // The five methods:
  public String getID() {
    return ID;
  }

  public boolean equals(BankAccount other) {
    return getID().equals(other.getID());
  }

  public double getBalance() {
    return balance;
  }

  public void deposit(double depositAmount) {
    balance = balance + depositAmount;
  }

  public void withdraw(double withdrawalAmount) {
    balance = balance - withdrawalAmount;
  }
}
```

This `BankAccount` type has been intentionally kept simple for ease of study. The available `BankAccount` messages include—but are not limited to—`withdraw`, `deposit`, `getID`, and `getBalance`. Each will store an account ID and a balance.

Instances of `BankAccount` are constructed with two arguments to help initialize these two values. You can supply two initial values in the following order:

1. a sequence of characters (a string) to represent the account identifier (a name, for example)
2. a number to represent the initial account balance

Here is one desired object construction that has two arguments for the purpose of initializing the two desired values:

```
BankAccount anAccount = new BankAccount("Chris", 125.50);
```

The construction of new objects (the creation of new instances) requires the keyword `new` with the class name and any required initial values between parentheses to help initialize the state of the object. The general form for creating an instance of a class:

***General Form:* Constructing objects (initial values are optional)**

class-name object-name = **new** *class-name*(*initial-value(s)*);

Every **object** has

1. a name (actually a reference variable that stores a reference to the object)
2. state (the set of values that the object remembers)
3. messages (the things objects can do and reveal)

Every instance of a class will have a reference variable to provide access to the object. Every instance of a class will have its own unique state. In addition, every instance of a class will understand the same set of messages. For example, given this object construction,

```
BankAccount anotherAccount = new BankAccount("Justin", 60.00);
```

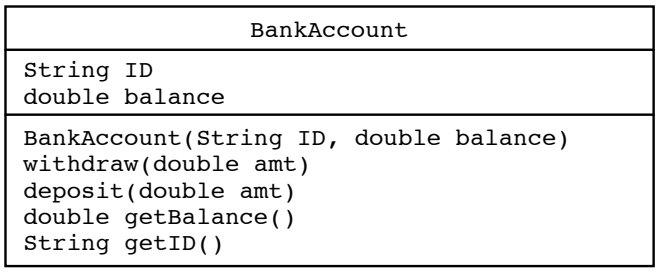
we can derive the following information:

1. name: `anotherAccount`
2. state: an account ID of `"Justin"` and a balance of `60.00`
3. messages: anotherAccount understands `withdraw`, `deposit`, `getBalance`, …

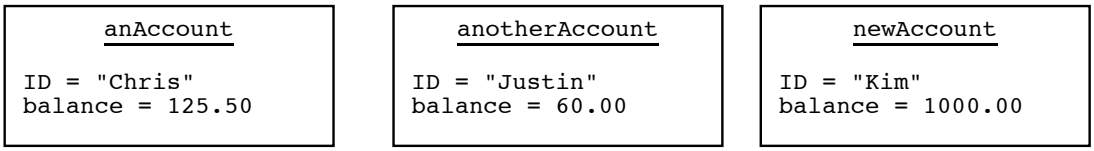
Other instances of `BankAccount` will understand the same set of messages. However, they will have their own separate state. For example, after another `BankAccount` construction,

```
BankAccount theNewAccount = new BankAccount("Kim", 1000.00);
```

`theNewAccount` has its own ID of "Kim" and its own balance of 1000.00. The three characteristics of an object can be summarized with diagrams. This class diagram represents one class.



A class diagram lists the class name in the topmost compartment. The instance variables appear in the compartment below it. The bottom compartment captures the methods. Objects can also be summarized in instance diagrams.



These three object diagrams describe the current state of three different `BankAccount` objects. One class can be used to make have many objects, each with its own separate state (set of values).

Sending Messages to Objects

In order for objects to do something, your code must send messages to them. A **message** is a request for the object to provide one of it services through a method.

General Form: **Sending a message to an object**

object-name`.`*message-name*`(`*argument1*`,` *argument2*`,` *…*`)`

Some messages ask for the state of the object. Other messages ask an object to do something, For example, each `BankAccount` object was designed to have the related operations `withdraw`,

`deposit`, `getBalance`, and `getID`. These messages ask the two different `BankAccount` objects to return information:

```
anAccount.getID();
anAccount.getBalance();
anotherAccount.getID();
anotherAccount.getBalance();
```

These messages ask two different `BankAccount` objects to do something:

```
anAccount.withdraw(40.00);
anAccount.deposit(100.00);
anotherAccount.withdraw(20.00);
anotherAccount.deposit(157.89);
```

The optional **arguments**—expressions between the parentheses—are the values required by the method to fulfill its responsibility. For example, `withdraw` needs to know how much money to withdraw. On the other hand, `getBalance` doesn't need any arguments to return the current balance of the `BankAccount` object. The output below indicates `deposit` and `withdraw` messages modify the account balances in an expected manner:

```
// Construct two objects and send messages to them.
public class ShowTwoBankAccountObjects {

  public static void main(String[] args) {

    BankAccount b1 = new BankAccount("Kim", 123.45);
    BankAccount b2 = new BankAccount("Chris", 500.00);

    System.out.println("Initial values");
    System.out.println(b1.getID() + ": " + b1.getBalance());
    System.out.println(b2.getID() + ": " + b2.getBalance());

    b1.deposit(222.22);
    b1.withdraw(20.00);
    b2.deposit(55.55);
    b2.withdraw(10.00);
    System.out.println();
    System.out.println("Value after deposit and withdraw messages");
    System.out.println(b1.getID() + ": " + b1.getBalance());
    System.out.println(b2.getID() + ": " + b2.getBalance());
  }
}
```

Output

```
Initial values
Kim: 123.45
Chris: 500.0

Value after deposit and withdraw messages
Kim: 325.67
Chris: 545.55
```

3.2 Making Assertions about Objects with JUnit

The `println` statements in the program above reveal the changing state of objects. However, in such examples, many lines can separate the output from the messages that affect the objects. This makes it a bit awkward to match up the expected result with the code that caused the changes. The current and changing state of objects can be observed and confirmed by making assertions. An assertion is a statement that can relay the current state of an object or convey the result of a message to an object. Assertions can be made with methods such `assertEquals`.

General Form: JUnit's assertEquals method for int and double values

```
assertEquals(int expected, int actual);
assertEquals(double expected, double actual, double errorTolerance);
```

Examples to assert integer expressions:

```
assertEquals(2, 5 / 2);
assertEquals(14, 39 % 25);
```

Examples to assert a floating point expression:

```
assertEquals(325.67, b1.getBalance(), 0.001);
assertEquals(545.55, b2.getBalance(), 0.001);
```

With *`assertEquals`*, an assertion will be true—or will "pass"—if the *expected* value equals the *actual* value. When comparing floating-point values, a third argument is needed to represent the error tolerance, which is the amount by which two real numbers may differ and still be equal. (Due to round off error, and the fact that numbers are stored in base 2 (binary) rather than in base 10 (decimal), two expressions that we consider “equal” may actually differ by a very small amount. This textbook will often use the very small error tolerance of 1e-14 or 0.00000000000001. This means that the following two numbers would be considered equal within 1e-14:

```
assertEquals(1.23456789012345, 1.23456789012346, 1e-14);
```

In contrast, these numbers are not considered equal when using an error factor of `1e-14`.

```
assertEquals(1.23456789012345, 1.23456789012347, 1e-14);
```

So using 1e14 ensures two values are equals to 13 decimal places, which is about as close as you can get. JUnit assertions allow us to place the expected value next to messages that reveal the actual state. This makes it easer to demonstrate the behavior of objects and to learn about new types. Later, you will see how assertions help in designing and testing your own Java classes, by making sure they have the correct behavior.
The `assertEquals` method is in the `Assert` class of `org.junit`. The `Assert` class needs to be imported (shown later) or `assertEquals` needs to be qualified (shown next).

```
// Construct two BankAccount objects
BankAccount anAccount = new BankAccount("Kim", 0.00);
BankAccount anotherAccount = new BankAccount("Chris", 500.00);

// These assertions pass
org.junit.Assert.assertEquals(0.00, anAccount.getBalance(), 1e-14);
org.junit.Assert.assertEquals("Kim", anAccount.getID());
org.junit.Assert.assertEquals("Chris", anotherAccount.getID());
org.junit.Assert.assertEquals(500.00, anotherAccount.getBalance(), 1e-14);

// Send messages to the BankAccount objects
anAccount.deposit(222.22);
anAccount.withdraw(20.00);
```

```
  anotherAccount.deposit(55.55);
  anotherAccount.withdraw(10.00);
  // These assertions pass
  org.junit.Assert.assertEquals(202.22, anAccount.getBalance(), 1e-14);
  org.junit.Assert.assertEquals(545.55, anotherAccount.getBalance(), 1e-14);
```

To make these assertions, you must have access to the JUnit testing framework, which is available in virtually all Java development environments. Eclipse does. Then assertions like those above can be placed in methods preceded by `@Test`. These methods are known as **test methods**. They are most often used to test a new method. The test methods here demonstrate some new types. A test method begins in a very specific manner:

```
  @org.junit.Test
  public void testSomething()  {   // more to come
```

Much like the `main` method, test methods are called from another program (JUnit). Test methods need things from the org.junit packages. This code uses `fully qualified names`.

```
public class FirstTest {

  @org.junit.Test // Marks this as a test method.
  public void testDeposit() {
    BankAccount anAccount = new BankAccount("Kim", 0.00);
    anAccount.deposit(123.45);
    org.junit.Assert.assertEquals(123.45, anAccount.getBalance(), 0.01);
  }
}
```

Adding imports shortens code in all test methods. This feature allows programmers to write the method name without the class to which the method belongs. The modified class shows that imports reduce the amount of code by `org.junit.Assert` and `ord.junit` for every test method and assertion, which is a good thing since much other code that is required.

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class FirstTest {

  @Test  // Marks this as a test method.
  public void testDeposit() {
    BankAccount anAccount = new BankAccount("Kim", 0.00);
    anAccount.deposit(123.45);
    assertEquals(123.45, anAccount.getBalance());
  }

  @Test  // Marks this as a test method.
  public void testWithdraw() {
    BankAccount anotherAccount = new BankAccount("Chris", 500.00);
    anotherAccount.withdraw(160.01);
    assertEquals(339.99, anotherAccount.getBalance());
  }
} // End unit test for BankAccount
```

Running JUnit

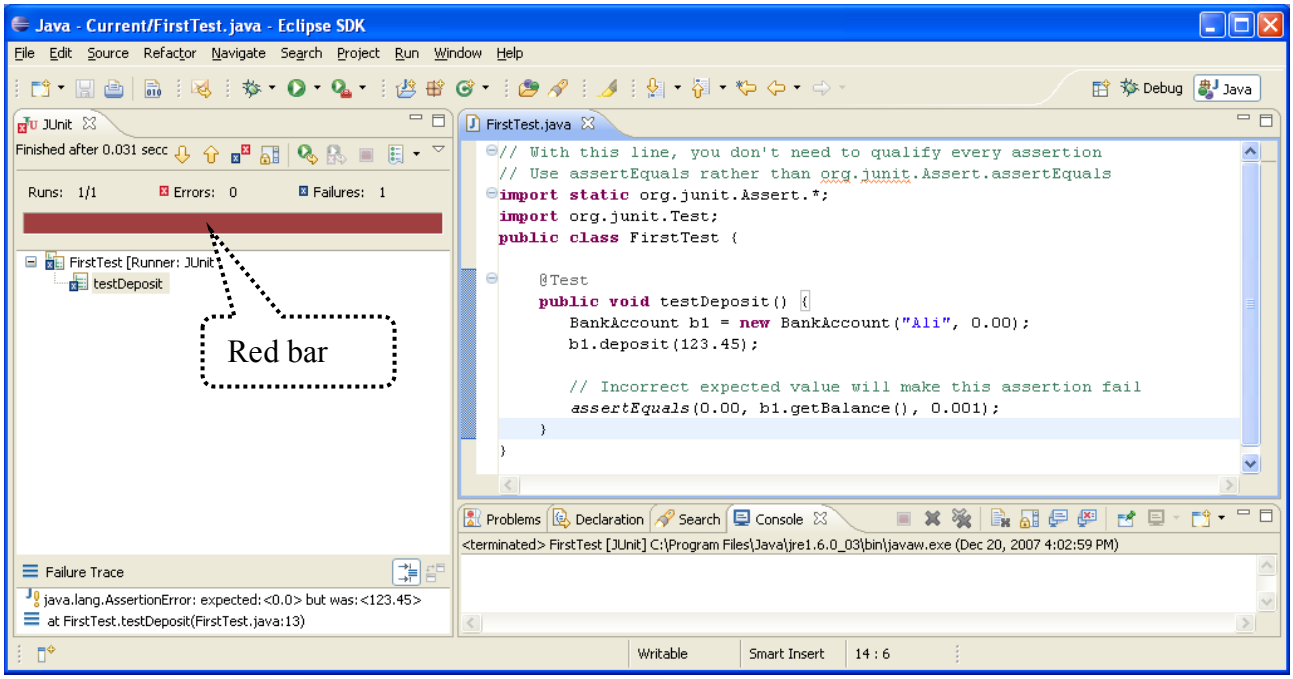
An assertion passes when the actual value equals the expected value in `assertEquals`.

```
  assertEquals(4, 9 / 2); // Assertion passes
```

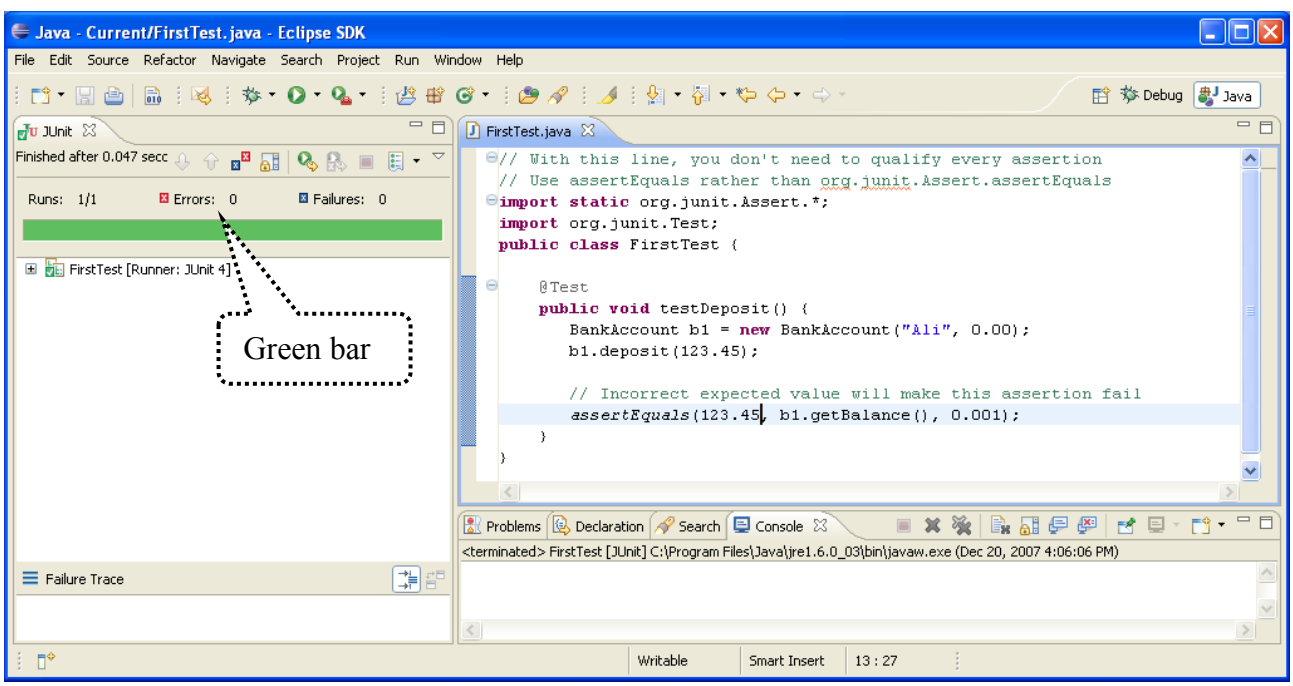
An assertion failes when the actual values does not equal the expected value.

```
assertEquals(4.5, 9 / 2, 1e-14); // Assertion fails
```

With integrated development environments such as Eclipse, Netbeans, Dr. Java, BlueJ, when an assertion fails, you see a red bar. For example, this screenshot of Eclipse shows a red bar.



The expected and actual values are shown in the lower left corner when the code in FirstTest.java is run as a JUnit test. Changing the `testDeposit` method to have the correct expected value results in a green bar, indicating all assertions have passed successfully. Here is JUnit's window when all assertions pass:



assertTrue and assertFalse

JUnit Assert class has several other methods to demonstrate and test code. The `assertTrue` assertion passes if its Boolean expression argument evaluates to true. The `assertFalse` assertion passes if the Boolean expression evaluates to false.

```
// Use two other Assert methods: assertTrue and assertFalse
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;
import org.junit.Test;

public class SecondTest {

  @Test
  public void showAssertTrue() {
    int quiz = 98;
    assertTrue(quiz >= 60);
  }

  @Test
  public void showAssertFalse() {
    int quiz = 55;
    assertFalse(quiz >= 60);
  }
}
```

The three Assert methods—`assertEquals`, `assertTrue`, and `assertFalse`—cover most of what we'll need.

3.3 String Objects

Java provides a `String` type to store a sequence of characters, which can represent an address or a name, for example. Sometimes a programmer is interested in the current length of a `String` (the number of characters). It might also be necessary to discover if a certain `substring` exists in a string. For example, is the substring `", "` included in the string `"Last, First"`. and if so, where does `substring ", "` begin? Java's `String` type, implemented as a Java class, provides a large number of methods to help with such problems required knowledge of the string value. You will use `String` objects in many programs.
Each `String` object stores a collection of zero or more characters. `String` objects can be constructed in two ways.

General Form: **Constructing `String` objects in two different ways**

`String` *identifier* **= `new String(`***string-literal***`);`**
`String` *identifier* **=** *string-literal***`;`**

Examples

```
String stringReference = new String("A String Object");
String anotherStringReference = "Another";
```

String length

For more specific examples, consider two `length` messages sent to two different `String` objects. Both messages evaluate to the number of characters in the `String`.

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;
public class StringTest {

  @Test
  public void showLength() {
    String stringReference = new String("A String Object");
    String anotherStringReference = "Another";
    // These assertions pass
    assertEquals(15, stringReference.length());
    assertEquals(7, anotherStringReference.length());
  }
}
```

String charAt

A `charAt` message returns the character located at the index passed as an `int` argument. Notice that `String` objects have zero-based indexing. The first character is located at index 0, and the second character is located at index 1, or `charAt(1)`.

```
  @Test
  public void showcharAt() {
    String stringReference = new String("A String");

    assertEquals('A', stringReference.charAt(0)); // Evaluates to 'A'
    assertEquals('r', stringReference.charAt(4)); // Evaluates to 'r'

    int len = stringReference.length() - 1;
    assertEquals('g', stringReference.charAt(len)); // The last char
  }
```

String indexOf

An `indexOf` message sent to a `String` object returns the index of the first character where the `String` argument is found. For example, `"no-yes".indexOf("yes")` returns 3. If the `String` argument does not exist, `indexOf` returns -1.

```
  @Test
  public void showIndexOf() {
    String stringReference = new String("A String Object");
    assertEquals(3, stringReference.indexOf("tri"));
    assertEquals(-1, stringReference.indexOf("not here"));
  }
```

Concatenation with the + operator

Programmers often make one `String` object from two separate strings with the + operator, that concatenates (connects) two or more strings into one string.

```
  @Test
  public void showConcatenate() {
    String firstName = "Kim";
    String lastName = "Madison";
    String fullName = lastName + ", " + firstName;
    assertEquals("Madison, Kim", fullName);
  }
```

String substring

A substring message returns the part of a string indexed by the beginning index through the ending index minus 1.

```
@Test
public void showSubString() {
  String str = "Smiles a Lot";
  assertEquals("mile", str.substring(1, 5));
}
```

String toUpperCase and toLowerCase

A `toUpperCase` message sent to a `String` object returns a new string that is the uppercase equivalent of the receiver of the message. A `toLowerCase` message returns a new string with all uppercase letters in lowercase.

```
@Test
public void testToUpperCase() {
  String str = new String("MiXeD cAsE!");
  assertEquals("MIXED CASE!", str.toUpperCase());
  assertEquals("mixed case!", str.toLowerCase());
  assertEquals("MiXeD cAsE!", str);  // str did not change!
}
```

Although it may sound like `toUpperCase` and `toLowerCase` modify String objects, they do not. Once constructed, String objects can not be changed. String objects are immutable. Simply put, there are no `String` messages that can modify the state of a `String` object. The final assertion above shows that `str.equals("MiXeD cAsE!")` still, even after the other two messages were sent. Strings are immutable to save memory. Java also supplies `StringBuilder`, a string type that has methods that do modify the objects.

Use an assignment if you want to change the String reference to refer to a different String.

```
@Test
public void showHowToUpperCaseWithAssignment() {
  String str = new String("MiXeD cAsE!");
  str = str.toUpperCase();
  assertEquals("MIXED CASE!", str); // str references a new string
}
```

Comparing Strings with `equals` and `compareTo`

JUnit's `assertEquals` method uses Java's `equals` method to compare the strings. This is the way to see if two String objects have the same sequence of characters. It is case sensitive.

```
@Test
public void showStringEquals() {
  String s1 = new String("Casey");
  String s2 = new String("Casey");
  String s3 = new String("CaSEy");
  assertTrue(s1.equals(s2));
  assertFalse(s1.equals(s3));
}
```

Never use == to compare string values. Always use the equals method!

Avoid using == to compare strings. The results can be surprising.

```
@Test
public void showCompareStringsWithEqualEqual() {
  String s1 = new String("Casey");
  assertTrue(s1 == "Casey");   // This assertion fails.
}
```

The == with objects compares reference values, not the actual values of the objects. The above code generates two different String objects that just happen to have the same state. Since the

reference values differ they are NOT equal using ==. Do not use ==. Instead use the `equals` method of `String`. The equals method was designed to compare the actual values of the string—the actual characters and lengths, not the reference values.

```
@Test
public void showCompareStringWithEquals() {
  String s1 = "Casey";
  assertTrue(s1.equals("Casey"));   // This assertion passes.
}
```

compareTo

For better or for worse, Java string objects can not be compared with relational operators such as > or <=. To test if one string is less than another, use `compareTo` messages that return a negative integer if the object before the dot is less than (alphabetically precedes) the string argument:

`"Casey".compareTo("Devon")` returns a negative integer

`compareTo` returns a positive integer if the object before the dot is greater than (alphabetically follows) the string argument:

`"Devon".compareTo("Casey")` returns a positive integer

`compareTo` returns zero if the object before the dot `equals` the string argument:

`"Casey".compareTo("Casey")` returns 0

The compareTo method needs to be used with a relational operator and **0** as shown in this `@Test` method:

```
@Test
public void showCompareStringWithEquals() {
  assertTrue("Casey".compareTo("Devon") < 0);
  assertTrue("Devon".compareTo("Casey") > 0);
  assertTrue("Casey".compareTo("Casey") == 0);
}
```

The `compareTo` method defines the natural ordering of objects. Many Java types have the `compareTo` method. The awkward thing about using `compareTo` is that it does not read as easily as "Casey" < "Devon".

Suggestion: while writing or reading `compareTo` messages, rather than doing it strictly from left to right, do it as object before the dot first ❶, skip over to the relational operator ❷, and lastly reverse to the argument to `compareTo` ❸

```
 ❶                ❸     ❷
"Casey".compareTo("Devon") < 0   // Is "Casey" < "Devon"?
```

Self-Check

3-1 Each of the lettered lines has an error. Explain why.

```
BankAccount b1 = new BankAccount("B.  ");          // a
BankAccount b2("The ID", 500.00);                  // b
BankAccount b3 = new Account("N.  Li", 200.00);    // c
b1.deposit();                                      // d
b1.deposit("100.00");                              // e
b1.Deposit(100.00);                                // f
withdraw(100);                                     // g
System.out.println(b4.getID());                    // h
```

3-2 What values makes these assertions pass (fill in the blanks)?

```
@Test public void testAcct() {
   BankAccount b1 = new BankAccount("Kim", 0.00);
   BankAccount b2 = new BankAccount("Chris", 500.00);
   assertEquals([        ], b1.getID());
   assertEquals([        ], b2.getID());
   b1.deposit(222.22);
   b1.withdraw(20.00);
   assertEquals([        ], b1.getBalance(), 0.001);
   b2.deposit(55.55);
   b2.withdraw(10.00);
   assertEquals([        ], b2.getBalance(), 0.001);
  }
}
```

3-3 What value makes this assertion pass?

```
String s1 = new String("abcdefghi");
assertEquals([       ], s1.indexOf("g"));
```

3-4 What value makes this assertion pass?

```
String s2 = "abcdefghi";
assertEquals([        ], s2.substring(4, 6));
```

3-5 Write an expression to store the middle character of a String into a `char` variable named `mid`. If there is an even number of characters, store the `char` to the right of the middle. For example, the middle character of "abcde" is `'c'` and of "Jude" is `'d'`.

3-6 For each of the following messages, if there is something wrong, write "error"; otherwise, write the value of the expression.

```
String str = new String("Any String");
```

a. `length(str)`
b. `str.length`
c. `str(length)`
d. `str.indexOf(" ")`
e. `str.substring(2, 5)`
f. `str.substring("tri")`

3-6 For each of the following messages, us the expression true or false?

```
String str = "def";
```

a. `str.compareTo("later") <= 0`
b. `str.compareTo("later") > 0`
c. `str.compareTo(s) == 0`
d. `str.compareTo(s) != 0`
e. `"abc".compareTo(str) >= 0`
f. `"abc".compareTo(str) < 0`
g. `"abc".compareTo("abcdef".substring(2, 4)) < 0`
h. `"abc".compareTo("abcdef".substring(2, 4)) > 0`

Answers to Self-Checks

3-1 -a Missing the second argument in the object construction. Add the starting balance—a number.
-b Missing `= new BankAccount`.
-c Change `Account` to `BankAccount`.
-d Missing a numeric argument between `(` and `)`.
-e Argument type wrong. pass a number, not a `String`.
-f `Deposit` is not a method of `BankAccount`. Change D to d.
-g Need an object and a dot before `withdraw`.
-h `b4` is not a `BankAccount` object. It was never declared to be anything.

3-2 a? "Kim"
b? "Chris"
c? 202.22
d? 545.55

3-3 6

3-4 `"ef"`

3-5
```
String aString = "abcde";
int midCharIndex = aString.length( ) / 2;
char mid = aString.charAt( midCharIndex );
```

3-6

-a	error	-d	3
-b	error	-e	`y s`
-c	error	-f	error (wrong type of argument)

3- 7
a. **`true`**
b. **`false`**
c. **`true`**
d. **`false`**
e. **`false`**
f. **`true`**
g. **`true`**
h. **`false`**

Chapter 4

Methods

Goal

- Implement well-tested Java methods

4.1 Methods

A java class typically has two or more methods. There are two major components to a method:

1. the method **heading**
2. the block (a pair of curly braces with code to complete the method's functionality

Several modifiers may begin a method heading, such as `public` or `private`. The examples shown here will use only the modifier `public`. Whereas **`private`** methods are only accessible from the class in which they exist, **`public`** methods are visible from other classes. Here is a general form for method headings.

***General Form:* A public method heading**

`public` *return-type method-name* **`(`***parameter-1***`,`** *parameter-2***`,`** *...,* *parameter-n* **`)`**

The *return-type* represents the type of value returned from the method. The return type can be any primitive type, such as `int` or `double` (as in `String`'s `length` method or `BankAccount`'s `withdraw` method, for example). Additionally, the return type can be any reference type, such as `String` or `Scanner`. The return type may also be `void` to indicate that the method returns nothing, as see in `void main` methods.

The *method-name* is any valid Java identifier. Since most methods need one or more values to get the job done, method headings may also specify **parameters** between the required parentheses. Here are a few syntactically correct method headings:

Example Method Headings

```
public int charAt(int index)                          // String
public void withdraw(double withdrawalAmount)         // BankAccount
public int length()                                   // String
public String substring(int startIndex, int endIndex) // String
```

The other part of a method is the body. A method body begins with a curly brace and ends with a curly brace. This is where the programmer places variable declarations, object constructions, assignments, and other messages that accomplish the purpose of the method. For example, here is the very simple `deposit` method from the `BankAccount` class. This method has access to the parameter `depositAmount` and to the `BankAccount` instance variable named `myBalance`

(instance variables are discussed in a later chapter).

```
// The method heading . . .
public void deposit(double depositAmount) {
  // followed by the method body
  myBalance = myBalance + depositAmount;
}
```

Parameters

A **parameter** is an identifier declared between the parentheses of a method heading. Parameters specify the number and type of arguments that must be used in a message. For example, `depositAmount` in the `deposit` method heading above is a parameter of type `double`. The programmer who wrote the method specified the number and type of values the method would need to do its job.

A method may need one, two, or even more arguments to accomplish its objectives. “How much money do you want to withdraw from the `BankAccount` object?” “What is the beginning and ending index of the `substring` you want?” “How many days do you want to add". Parameters provide the mechanism to get the appropriate information to the method when it is called. For example, a `deposit` message to a `BankAccount` object requires that the amount to be deposited, (a `double`), be supplied.

```
public void deposit(double depositAmount)
                                ↑
        anAccount.deposit(123.45);
```

When this message is sent to anAccount, the value of the argument 123.45 is passed on to the associated parameter `depositAmount`. It may help to read the arrow as an assignment statement. The argument `123.45` is assigned to `depositAmount` and used inside the `deposit` method. This example has a literal argument `(123.45)`. The argument may be any expression that evaluates to the parameter’s declared type, such as `(checks + cash)`.

```
double checks = 123.45;
double cash = 100.00;
anAccount.deposit(checks + cash);
```

When there is more than one parameter, the arguments are assigned in order. The replace method of the `String` type requires two character values so the method knows which character to replace and with which character.

```
public String replace(char oldChar, char newChar)
                                ↖       ↗
  String newString = str.replace('t', 'X');
```

Reading Method Headings

When properly documented, the first part of a method, the heading, explains what the method does and describe the number of arguments and the type All of these things allow the programmer to send messages to objects without knowing the details of the implementation of those methods. For example, to send a message to an object, the programmer must:

- know the method name
- supply the proper number and type of arguments
- use the return value of the method correctly

All of this information is specified in the method heading. For example, the `substring` method of Java's `String` class takes two `int` arguments and evaluates to a `String`.

```
// Return portion of this string indexed from beginIndex through endIndex-1
public String substring(int beginIndex, int endIndex)
```

The method heading for `substring` provides the following information:

- type of value returned by the method: `String`
- method name: `substring`
- number of arguments required: 2
- type of the arguments required: both are `int`

Since `substring` is a method of the `String` class, the message begins with a reference to a string before the dot.

```
String str = new String("small");
assertEquals("mall", str.substring(1, str.length()));

// Can send messages to String literals ...
assertEquals("for", "forever".substring(0, 3));
```

A `substring` message requires two arguments, which specify the beginning and ending index of the `String` to return. This can be observed in the method heading below, which has two parameters named `beginIndex` and `endIndex`. Both arguments in the message `fullName.substring(0,6)` are of type `int` because the parameters in the `substring` method heading are declared as type `int`.

```
public String substring(int beginIndex, int endIndex)
                                  ↑           ↑
                fullName.substring(0,         6);
```

When this message is sent, the argument 0 is assigned to the parameter `beginIndex`, and the argument 6 is assigned to the parameter `endIndex`. Control is then transferred to the method body where this information is used to return what the method promises. In general, when a method requires more than one argument, the first argument in the message will be assigned to the first parameter, the second argument will be assigned to the second parameter, and so on. In order to get correct results, the programmer must also order the arguments correctly. Whereas not supplying the correct number and type of arguments in a message results in a compile time (syntax) error, supplying the correct number and type of arguments in the wrong order results in a logic error (i.e., the program does what you typed, not what you intended).
And finally, there are several times when the `substring` method will throw an exception because the integer arguments are not in the correct range.

```
String str = "abc";
str.substring(-1, 1)  // Runtime error because beginIndex < 0
str.substring(0, 4)   // Runtime error because endIndex of 4 is off by 1
str.substring(2, 1)   // Runtime error because beginIndex > endIndex
```

Self-Check

Use the following method heading to answer the first three questions that follow. This `concat` method is from Java's `String` class.

```
// Return the concatenation of str at the end of this String object
public String concat(String str)
```

4-1 Using the method heading above, determine the following for String's concat method:

-a return type
-b method name
-c number of arguments
-d first argument type (or class)
-e second argument type (or class)

4-2 Assuming `String s = new String("abc");`, write the return value for each valid message or explain why the message is invalid.

```
-a s.concat("xyz");   -d s.concat("x", "y");
-b s.concat();        -e s.concat("wx" + " yz");
-c s.concat(5);       -f s.concat("d");
```

4-3 What values make these assertions pass?

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class StringTest {

  @Test
  public void testConcat() {
     String s = "abc";
     assertEquals([          ], s.concat("!"));
     assertEquals([          ], s.concat("cba"));
     assertEquals([          ], s.concat("123"));
  }
}
```

Use the following method heading to answer the first three questions that follow. This `concat` method is from Java's `String` class.

```
// Returns a new string resulting from replacing all
// occurrences of oldChar in this string with newChar.
public String replace(char oldChar, char newChar)
```

4-4 Using the method heading above, determine the following for String's replace method:

-a return type
-b method name
-c number of arguments
-d first argument type
-e second argument type

4-5 Assuming `String s = new String("abcabc");`, write the return value for each valid message or explain why the message is invalid.

```
-a s.replace("a");          -d s.replace("x", "y");
-b s.replace('c', 'Z');     -e s.replace('a', 'X');
-c s.replace('b', 'Z');     -f s.concat('X', 'a');
```

4-6 What values make the assertions pass?

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class StringTest {
  @Test
  public void testReplace () {
     String s = "aabbcc";
     assertEquals("__a.___", s.replace('a', 'T'));
     assertEquals("__b.___", s.replace ('b', ' '));
     assertEquals("__c.___", s.replace ('c', 'Y'));
  }
}
```

Methods that return Values

When a method is called, the values of the arguments are copied to the parameters so the values can be used by the method. The flow of control then transfers to the called method where those statements are executed. One of those statements in all non-void methods must return a value. This is done with the Java `return` statement that allows a method to return information. Here is the general form:

General Form **return statement**

`return` *expression*`;`

The following examples show the return statement in the context of complete methods. The three methods are captured in a class named `ExampleMethods`, which implies there is no relationship between the methods. It simply provides methods with different return types.

```
// This class contains several unrelated methods to provide examples.
public class ExampleMethods {

  // Return a number that is twice the value of the argument.
  public double f(double argument) {
    return 2.0 * argument;
  }

  // Return true if argument is an odd integer, false when argument is even.
  public boolean isOdd(int argument) {
    return (argument % 2 != 0);
  }

  // Return the first two and last two characters of the string.
  // Precondition: str.length() >= 4
  public String firstAndLast(String str) {
    int len = str.length();
    String firstTwo = str.substring(0, 2);
    String lastTwo = str.substring(len - 2, len);
    return firstTwo + lastTwo;
  }

} // End of class with three example methods.
```

When a `return` statement is encountered, the *expression* that follows `return` replaces the message part of the statement. This allows a method to communicate information back to the caller. Whereas a `void` method returns nothing (see any of the void `main` methods or `test` methods), any method that has a return type other than `void` *must* return a value that

matches the return type. So, a method declared to return a `String` must return a reference to a `String` object. A method declared to return a `double` must return a primitive `double` value. Fortunately, the compiler will complain if you forget to return a value or you attempt to return the wrong type of value.

As suggested earlier, testing can occur at many times during software development. When you write a method, test it. For example, a test method for `firstAndLast` could look like this.

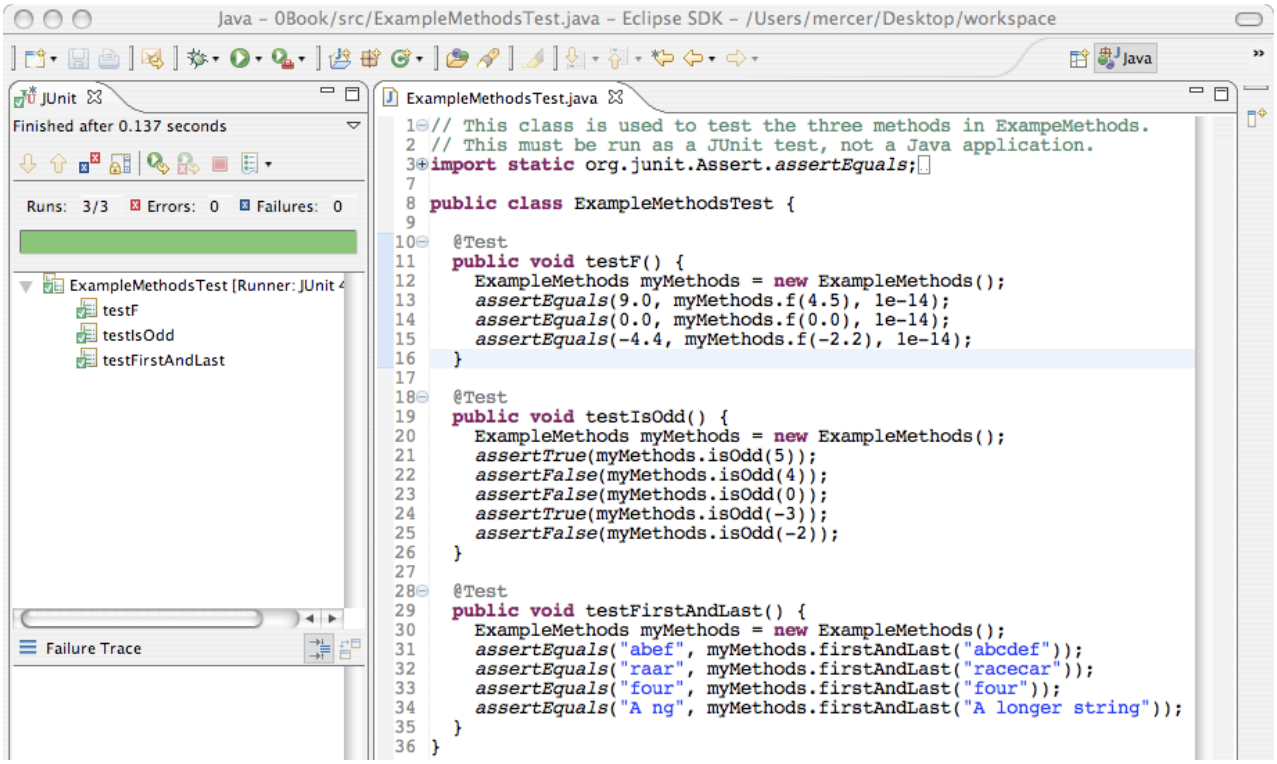
```
  @Test
  public void testFirstAndLast() {
    ExampleMethods myMethods = new ExampleMethods();
    assertEquals("abef", myMethods.firstAndLast("abcdef"));
    assertEquals("raar", myMethods.firstAndLast("racecar"));
    assertEquals("four", myMethods.firstAndLast("four"));
    assertEquals("A ng", myMethods.firstAndLast("A longer string"));
  }
```

Methods may exist in any class. We could use test methods in the same class as the methods being tested because it is convenient to write methods and tests in the same file. That approach would also have the benefit not requiring an `new ExampleMethods()` object thereby requiring us to write less code. However, it is common practice to write tests in a separate test class. Conveniently, we can place test methods for each of the three `ExampleMethods` in another file keeping tests separate from the methods.

```
// This class is used to test the three methods in ExampleMethods.
import static org.junit.Assert.*;
import org.junit.Test;

public class ExampleMethodsTest {

  @Test
  public void testF() {
    ExampleMethods myMethods = new ExampleMethods();
    assertEquals(9.0, myMethods.f(4.5), 1e-14);
    assertEquals(0.0, myMethods.f(0.0), 1e-14);
    assertEquals(-4.4, myMethods.f(-2.2), 1e-14);
  }

  @Test
  public void testIsOdd() {
    ExampleMethods myMethods = new ExampleMethods();
    assertTrue(myMethods.isOdd(5));
    assertFalse(myMethods.isOdd(4));
    assertFalse(myMethods.isOdd(0));
    assertTrue(myMethods.isOdd(-3));
    assertFalse(myMethods.isOdd(-2));
  }

  @Test
  public void testFirstAndLast() {
    ExampleMethods myMethods = new ExampleMethods();
    assertEquals("abef", myMethods.firstAndLast("abcdef"));
    assertEquals("raar", myMethods.firstAndLast("racecar"));
    assertEquals("four", myMethods.firstAndLast("four"));
    assertEquals("A ng", myMethods.firstAndLast("A longer string"));
  }
}
```

This way to implement and test methods IS made possible with the JUnit testing framework. Most college textbooks use printlns and user input to show the results of running code that requires several program runs with careful input of values and careful inspection of the output

each time. This textbook integrates testing with JUnit, an industry-level testing framework that makes software development more efficient and less error prone. It is easer to test and debug your code. You are more likely to find errors more quickly. When run as a JUnit test, all assertions pass in all three test-methods and the green bar appears.



With JUnit, you can set up your tests and methods and run them with no user input. The process can be easily repeated while you debug. Writing assertions also makes us think about what the method should do before writing the method. Writing assertions will help you determine how to best test code now and into the future, a worthwhile skill to develop that costs little time.

Self-Check

4-7 a) Write a complete test method named `testInRange` as if it were in class `ExampleMethodsTest` to test method `inRange` that will be placed in class `ExampleMethods`. Here is the method heading for the method that will go into class `ExampleMethods`.

```
// Return true if number is in the range of 1 through 10 inclusive.
public boolean inRange(int number)
```

b) Write the complete method named `inRange` as if it were in `ExampleMethods`.

4-8 a) Write a complete test method named `testAverageOfThree` as if it were in class `ExampleMethodsTest` to test method `averageOfThree` that will be placed in class `ExampleMethods`. Here is the method heading for the method that will go into class `ExampleMethods`.

```
// Return the average of the three arguments.
public double averageOfThree(double a, double b, double c)
```

b) Write the complete method named `averageOfThree` as if it were in `ExampleMethods`.

4-9 a) Write a complete test method named `testRemoveMiddleTwo` as if it were in class `ExampleMethodsTest` to test method `removeMiddleTwo` that will be placed in class `ExampleMethods`. `removeMiddleTwo` should return a string that has all characters except the two in the middle. Assume the String argument has two or more characters. Here is the method heading for the method that will go into class `ExampleMethods`.

```
// Return the String argument with the middle two character missing.
// removeMiddleTwo("abcd") should return "ad"
// removeMiddleTwo("abcde") should return "abd"
// Precondition: sr.length() >= 2
public String removeMiddleTwo(String str)
```

b) Write the complete method named `removeMiddleTwo` as if it were in the `ExampleMethods` class.

How do we know what to test?

Methods are deigned to have parameters to allow different arguments. This makes them generally useful in future applications. But how do we know these methods work? Is it important that they are correct? Software quality is important. It is impossible to write perfect code.

One effective technique to ensure a method does what it is supposed to do is to write assertions to fully test the method. Asserting a method returns the correct value for one value is usually not enough. How many assertions should we make? What arguments should we use? The answers are not preordained. However, by pushing the limits of all the possible assertions and values we can think of, and doing this repeatedly, we get better at testing. Examples help. Consider this maxOfThree method.

```
// Return the maximum value of the integer arguments.
public int maxOfThree(int a, int b, int c)
```

As recommended in Chapter 1, it helps to have sample input with the expected result. Some test cases to consider include all three numbers the same, all 0, and certainly all different. Testing experts will tell you that test cases include all permutations of the different integers. So the test cases should include the max of (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), and (3, 2, 1). Whenever negative numbers are allowed, write assertions with negative numbers.

This large number of test cases probably seems excessive, but it doesn't take much time. There are a large number of algorithms that will make maxOfThree work. I have personally seen many of these that work in most cases, but not all cases. Especially interesting are the test cases when two are equal (students often write > rather than >=). So other test cases should include the max of (1, 2, 2), (2, 1, 2), and (2, 2, 1).

Since we can setup these test cases with the expected value and actual value next to each other and then run the tests once (or more than once if you detect a bug or use incorrect expected values). This test method contains more assertions than you would typically need due to the nature of the problem where the largest could be any of the three arguments and any one could equal another two.

```
@Test
public void testMaxOfThree() {
  ExampleMethods myMethods = new ExampleMethods();
  // All equal
  assertEquals(5, myMethods.maxOfThree(5, 5, 5));
  assertEquals(-5, myMethods.maxOfThree(-5, -5, -5));
  assertEquals(0, myMethods.maxOfThree(0, 0, 0));
  // All permutations of 3 different arguments
  assertEquals(3, myMethods.maxOfThree(1, 2, 3));
  assertEquals(3, myMethods.maxOfThree(1, 3, 2));
```

```
    assertEquals(3, myMethods.maxOfThree(2, 1, 3));
    assertEquals(3, myMethods.maxOfThree(2, 3, 1));
    assertEquals(3, myMethods.maxOfThree(3, 1, 2));
    assertEquals(3, myMethods.maxOfThree(3, 2, 1));

    // All permutations of two integers that are the largest
    assertEquals(2, myMethods.maxOfThree(1, 2, 2));
    assertEquals(2, myMethods.maxOfThree(2, 1, 2));
    assertEquals(2, myMethods.maxOfThree(2, 2, 1));

    // All permutations of two integers that are the smallest
    assertEquals(2, myMethods.maxOfThree(1, 1, 2));
    assertEquals(2, myMethods.maxOfThree(2, 1, 1));
    assertEquals(2, myMethods.maxOfThree(1, 2, 1));
  }
```

Self-Check

4-10 Consider a method that takes three integer arguments representing the three sides of a triangle. The method must reports whether the triangle is scalene (three sides different), isosceles (two sides equal), equilateral, or not a triangle (cannot be). What tests should be written and for each, what should the result be?

4-11 Boggle tests your ability to find words in a random array of dice with letters. Words must be in the range of 3..16 characters inclusive. Method `inRange(String str)` must return true if the length of `str` is in the range of 3 through 16 characters inclusive. What tests should be written and for each, what should the result be?

Answers to Self-Check Questions

4-1
- -a String
- -b concat
- -c 1
- -d String
- -e There is no second parameter

4-2
- -a "abcxyz
- -b needs argument
- -c 5 wrong type;
- -d One too many arguments
- -e "abcwx yz"
- -f "abcd"

4-3
```
assertEquals("abc!", s.concat("!"));
assertEquals("abccba" , s.concat("cba"));
assertEquals("abc123", s.concat("123"));
```

4-4
- -a String
- -b replace
- -c 2
- -d char
- -e char

4-5
- -a need 2 char arguments
- -b "abZabZ"
- -c "aZcaZb
- -d wrong type arguments. Need char, not String
- -e "XbcXbc"
- -f Wrong type and number of arguments for concat

4-6
```
assertEquals("TTbbcc", s.replace('a', 'T'));
assertEquals("aa  cc", s.replace('b', ' '));
assertEquals("aabbYY", s.replace('c', 'Y'));
```

4-7 a)
```
@Test
public void testInRange() {
  assertFalse(inRange(0));
  assertTrue(inRange(1));
  assertTrue(inRange(5));
  assertTrue(inRange(10));
  assertFalse(inRange(11));
}
```

b)
```
public boolean inRange(int number) {
  return (number >= 1) && (number <= 10);
}
```

4-8 a)
```
@Test
public void testAverageThree() {
  ExampleMethods myMethods = new ExampleMethods();
  assertEquals(0.0, myMethods.averageOfThree(0.0, 0.0, 0.0), 0.1);
  assertEquals(90.0, myMethods.averageOfThree(90.0, 90.0, 90.0), 0.1);
  assertEquals(82.5, myMethods.averageOfThree(90.0, 80.5, 77.0), 0.1);
  assertEquals(-2.0, myMethods.averageOfThree(-1, -2, -3), 0.1);
}
```

b)
```
public double averageOfThree(double a, double b, double c) {
  return (a + b + c) / 3.0;
}
```

4-9 a)
```
@Test
public void testRemoveMiddleTwo() {
  ExampleMethods myMethods = new ExampleMethods();
  assertEquals("", myMethods.removeMiddleTwo("12"));
  assertEquals("ad", myMethods.removeMiddleTwo("abcd"));
  assertEquals("ade", myMethods.removeMiddleTwo("abcde"));
  assertEquals("abef", myMethods.removeMiddleTwo("abcdef"));
}
```

b)
```
public String removeMiddleTwo(String str) {
  int mid = str.length() / 2;
  return str.substring(0, mid-1) + str.substring(mid + 1, str.length());
}
```

4-10 Equilateral: (5, 5, 5)
Isosceles with permutations: (3, 3, 2) (2, 3, 3) (3, 2, 3)
Scalene with permutations: (2, 3, 4) (2 ,4, 3) (3, 2, 4) (3, 4, 2) (4 , 2, 3) (4, 3, 2)
Not a triangle and permutations: (1, 2, 3) (1, 3, 2) (2, 1, 3) (2, 3, 1) (3, 2, 1) (3, 1, 2)
Not a triangle and permutations: (1, 2, 4) (1, 4, 2) (2, 1, 4) (2, 4, 1) (4, 2, 1) (4, 1, 2)
Not a triangle and permutations: (0, 2, 3) (1, 0, 2) (2, 1, 0)
Not a triangle with negative lengths and permutations: (-1, 2, 3) (1, -2, 3) (1, 2, -3)
Not a triangle, all negative, but would be if equilateral if positive: (-5, -5, -5)

4-11
```
@Test
public void testInRangeString() {
  ExampleMethods myMethods = new ExampleMethods();
  assertFalse(myMethods.inRange("")); // Empty string
  assertFalse(myMethods.inRange("ab")); // On the border -1
  assertTrue(myMethods.inRange("abc")); // On the border
  assertTrue(myMethods.inRange("abcd")); // On the border + 1
  assertTrue(myMethods.inRange("abcdef")); // In the middle
  assertTrue(myMethods.inRange("1234567890")); // In the middle
  assertTrue(myMethods.inRange("123456789012345")); // On the border - 1
  assertTrue(myMethods.inRange("1234567890123456")); // On the border
  assertFalse(myMethods.inRange("12345678901234567")); // On the border + 1
}
```

Chapter 5
Selection

Goals

It is sometimes appropriate for certain actions to execute one time but not at other times. Sometimes the specific code that executes must be chosen from many alternatives. This chapter presents statements that allow such selections. After studying this chapter, you will be able to:

- see how Java implements the Guarded Action pattern with the `if` statement
- implement the Alternative Action pattern with the Java `if else`
- implement the Multiple Selection pattern with nested the `if else` statement

5.1 Selection

Programs must often anticipate a variety of situations. For example, an automated teller machine (ATM) must serve valid bank customers, but it must also reject invalid access attempts. Once validated, a customer may wish to perform a balance query, a cash withdrawal, or a deposit. The code that controls an ATM must permit these different requests. Without selective forms of control—the statements covered in this chapter—all bank customers could perform only one particular transaction. Worse, invalid PINs could not be rejected!

Before any ATM becomes operational, programmers must implement code that anticipates all possible transactions. The code must turn away customers with invalid PINs. The code must prevent invalid transactions such as cash withdrawal amounts that are not in the proper increment (of 10.00 or 20.00, for instance). The code must be able to deal with customers who attempt to withdraw more than they have. To accomplish these tasks, a new form of control is needed—a way to permit or prevent execution of certain statements depending on the current state.

The Guarded Action Pattern

Programs often need actions that do not always execute. At one moment, a particular action must occur. At some other time—the next day or the next millisecond perhaps—the same action must be skipped. For example, one student may make the dean's list because the student's grade point average (GPA) is 3.5 or higher. That student becomes part of the dean's list. The next student may have a GPA lower than 3.5 and should not become part of the dean's list. The action—adding a student to the dean's list—is guarded.

Algorithmic Pattern 5.1

Pattern:	Guarded Action
Problem:	Do something only if certain conditions are true.
Outline:	if (true-or-false-condition is true) execute this action
Code Example:	`if (GPA >= 3.5)` `  System.out.println("Made the dean's list");`

The `if` Statement

This Guarded Action pattern occurs so frequently it is implemented in most programming languages with the `if` statement.

***General Form:* `if` statement**

`if` **(***Boolean-expression)*
true-part

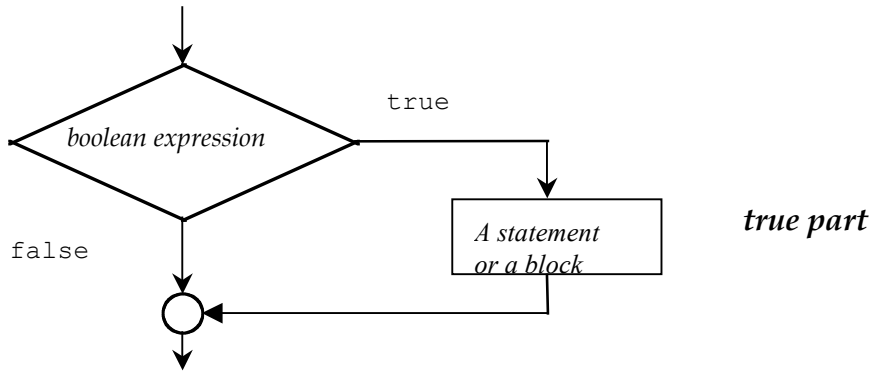
A *Boolean-expression* is any expression that evaluates to either true or false. The *true-part* may be any valid Java statement, including a block. A block is a sequence of statements within the braces `{` and `}`.

Examples of `if` Statements

```
if (hoursStudied > 4.5)
  System.out.println("You are ready for the test");

if (hoursWorked > 40.0) {
  // With a block with { } for the true part so both statements may execute
  regularHours = 40.0;
  overtimeHours = hoursWorked - 40.0;
}
```

When an `if` statement is encountered, the boolean expression is evaluated to false or true. The "true part" executes only if the boolean expression evaluates to true. So in the first example above, the output `"You are ready for the test"` appears only when the user enters something greater than 4.5. When the input is 4.5 or less, the true part is skipped—the action is guarded. Here is a flowchart view of the Guarded Action pattern:

Flowchart view of the Guarded Action pattern



A test method for `withdraw` illustrates that a `BankAccount` object should not change for negative arguments.

```
@Test
public void testGetWithdrawWhenNotPositive() {
  BankAccount anAcct = new BankAccount("Angel", 100.00);
  // Can't withdraw amounts <= 0.0;
  anAcct.withdraw(0.00);
  // Balance remains the same
  assertEquals(100.00, anAcct.getBalance(), 0.1);
  anAcct.withdraw(-0.99);
  // Balance remains the same
  assertEquals(100.00, anAcct.getBalance(), 0.1);
}
```

Nor should any `BankAccount` object change when the amount is greater than the balance.

```
@Test
public void testGetWithdrawWhenNotEnoughMoney() {
  BankAccount anAcct = new BankAccount("Angel", 100.00);
  // Do not want withdrawals when the amount > balance;
  anAcct.withdraw(100.01);  // Balance should remain the same
  assertEquals(100.00, anAcct.getBalance(), 0.1);
}
```

The if statement in this modified `wthdraw` method guards against changing the balance—an instance variable—when the argument is negative or greater than the balance

```
public void withdraw(double withdrawalAmount) {
  if (withdrawalAmount > 0.00 && withdrawalAmount <= balance) {
    balance = balance - withdrawalAmount;
  }
}
```

Through the power of the `if` statement, the same exact code results in two different actions. The `if` statement controls execution because the true part executes only when the Boolean expression is true. The `if` statement also controls statement execution by disregarding statements when the Boolean expression is false.

Self-Check

5-1 Write the output generated by the following pieces of code:

-a
```
int grade = 45;
if(grade >= 70)
  System.out.println("passing");
if(grade < 70)
  System.out.println("dubious");
if(grade < 60)
  System.out.println("failing");
```

-b
```
int grade = 65;
if( grade >= 70 )
  System.out.println("passing");
if( grade < 70 )
  System.out.println("dubious");
if( grade < 60 )
  System.out.println("failing");
```

-c
```
String option = "D";
if(option.equals("A"))
  System.out.println( "addRecord" );
if(option.equals("D"))
  System.out.println("deleteRecord")
```

5.2 The Alternative Action Pattern

Programs must often select from a variety of actions. For example, say one student passes with a final grade that is ≥ 60.0. The next student fails with a final grade that is < 60.0. This example uses the Alternative Action algorithmic pattern. The program must choose one course of action or an alternative.

Algorithmic Pattern: Alternate Action

Pattern:	Alternative Action
Problem:	Need to choose one action from two alternatives.
Outline:	if (true-or-false-condition is true) execute action-1 otherwise execute action-2

Code Example:

```
if(finalGrade >= 60.0)
   System.out.println("passing");
else
   System.out.println("failing");
```

The `if else` Statement

The Alternative Action pattern can be implemented with Java's **`if else`** statement. This control structure can be used to choose between two different courses of action (and, as shown later, to choose between more than two alternatives).

***The* `if else` Statement**

`if(`*boolean-expression)*
  true-part
`else`
  false-part

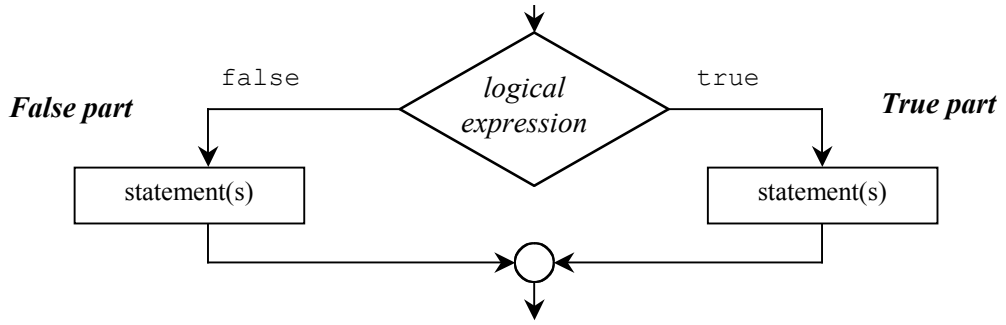
The `if else` statement is an `if` statement followed by the alternate path after an `else`. The *true-part* and the *false-part* may be any valid Java statements or blocks (statements and variable declarations between the curly braces `{` and `}`).

Example of if else Statements

```
if (sales <= 20000.00)
  System.out.println("No bonus");
else
  System.out.println("Bonus coming");


if (withdrawalAmount <= myAcct.getBalance()) {
  myAcct.withdraw(withdrawalAmount);
  System.out.println("Current balance: " + myAcct.getBalance());
}
else {
  System.out.println("Insufficient funds");
}
```

When an `if else` statement is encountered, the Boolean expression evaluates to either `false` or `true`. When `true`, the true part executes—the false part does not. When the Boolean expression evaluates to `false`, only the false part executes.

Flowchart view of the Alternative Action pattern



Self-Check

5-2 Write the output generated by each code segment given these initializations of `j` and `x`:

```
int j = 8;
double x = -1.5;
```

-a
```
if(x < -1.0)
  System.out.println("true");
else
  System.out.println("false");
  System.out.println("after if...else");
```

-b
```
if(j >= 0)
  System.out.println("zero or pos");
else
  System.out.println("neg");
```

-c
```
if(x >= j)
  System.out.println("x is high");
else
  System.out.println("x is low");
```

-d
```
if(x <= 0.0)
  if(x < 0.0) // True part is another if...else
    System.out.println("neg");
  else
    System.out.println("zero");
else
  System.out.println("pos");
```

5-3 Write an `if else` statement that displays your name if `int option` is an odd integer or displays your school if `option` is even.

A Block with Selection Structures

The special symbols `{` and `}` have been used to gather a set of statements and variable declarations that are treated as one statement for the body of a method. These two special symbols delimit (mark the boundaries) of a block. The block groups together many actions, which can then be treated as one. The block is also useful for combining more than one action as the true or false part of an `if else` statement. Here is an example:

```
double GPA;
double margin;
// Determine the distance from the dean's list cut-off
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter GPA: ");
GPA = keyboard.nextDouble();

if(GPA >= 3.5) {
  // True part contains more than one statement in this block
  System.out.println("Congratulations, you are on the dean's list.");
  margin = GPA - 3.5;
  System.out.println("You made it by " + margin + " points.");
}
else {
  // False part contains more than one statement in this block
  System.out.println("Sorry, you are not on the dean's list.");
  margin = 3.5 - GPA;
  System.out.println("You missed it by " + margin + " points.");
}
```

The block makes it possible to treat many statements as one. When `GPA` is input as `3.7`, the Boolean expression (`GPA >= 3.5`) is true and the following output is generated:

Dialog

```
Enter GPA: 3.7
Congratulations, you are on the dean's list.
You made it by 0.2 points.
```

When `GPA` is `2.9`, the Boolean expression (`GPA >= 3.5`) is false and this output occurs:

Dialog

```
Enter GPA: 2.9
Sorry, you are not on the dean's list.
You missed it by 0.6 points.
```

This alternate execution is provided by the two possible evaluations of the boolean expression `GPA >= 3.5`. If it evaluates to true, the true part executes; if false, the false part executes.

The Trouble in Forgetting `{` and `}`

Neglecting to use a block with `if else` statements can cause a variety of errors. Modifying the previous example illustrates what can go wrong if a block is not used when attempting to execute both output statements.

```
if(GPA >= 3.5)
  margin = GPA - 3.5;
  System.out.println("Congratulations, you are on the dean's list.");
  System.out.println("You made it by " + margin + " points.");
else  // <- ERROR: Unexpected else
```

With `{` and `}` removed, there is no block; the two bolded statements no longer belong to the preceding `if else`, even though the indentation might make it appear as such. This previous code represents an `if` statement followed by two `println` statements followed by the reserved word `else`. When `else` is encountered, the Java compiler complains because there is no statement that begins with an `else`.

Here is another example of what can go wrong when a block is omitted. This time, `{` and `}` are omitted after `else`.

```
else
  margin = 3.5 - GPA;
  System.out.println("Sorry, you are not on the dean's list.");
  System.out.println("You missed it by " + margin + " points.");
```

There are no compiletime errors here, but the code does contain an intent error. The final two statements always execute! They do not belong to `if else`. If `GPA >= 3.5` is false, the code does execute as one would expect. But when this boolean expression is true, the output is not what is intended. Instead, this rather confusing output shows up:

```
Congratulations, you are on the dean's list.
You made it by 0.152 points.
Sorry, you are not on the dean's list.
You missed it by -0.152 points.
```

Although it is not necessary, always using blocks for the true and false parts of `if` and `if else` statements could help you. The practice can make for code that is more readable. At the same time, it could help to prevent intent errors such as the one above. One of the drawbacks is that there are more lines of code and more sets of curly braces to line up. In addition, the action is often only one statement and the block is not required.

5.3 Multiple Selection

"Multiple selection" refers to times when programmers need to select one action from many possible actions. This pattern is summarized as follows:

Algorithmic Pattern: Multiple Selection

Pattern:	Multiple Selection
Problem:	Must execute one set of actions from three or more alternatives.

Outline:	if (condition-1 is true)
	  execute action-1
	else if(condition-2 is true)
	  execute action-2
	else if(condition n-1 is true)
	  execute action n-1
	else
	  execute action-n

Code Example:

```
// Return a message related to the "comfyness"
// of the size of the string argument
public String comfy(String str) {
  String result = "?";
  int size = str.length();

  if (size < 2)
    result = "Way too small";
  else if (size < 4)
    result = "Too small";
  else if (size == 4)
    result = "Just right";
  else if (size > 4 && size <= 8)
    result = "Too big";
  else
    result = "Way too big";

  return result;
}
```

The following code contains an instance of the Multiple Selection pattern. It selects from one of three possible actions. Any grade point average (GPA) less than 3.5 (including negative numbers) generates the output "Try harder." Any GPA less than 4.0 but greater than or equal to 3.5 generates the output "You made the dean's list." And any GPA greater than or equal to 4.0 generates the output "You made the president's list." There is no upper range or lower range defined in this problem.

```
// Multiple selection, where exactly one println statement
// executes no matter what value is entered for GPA.
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter your GPA: ");
double GPA = keyboard.nextDouble();
if (GPA < 3.5)
  System.out.println("Try harder");
else {
  // This false part of this if else is another if else
  if (GPA < 4.0)
    System.out.println("You made the dean's list");
  else
    System.out.println("You made the president's list");
}
```

Notice that the false part of the first `if else` statement is another `if else` statement. If `GPA` is less than 3.5, `Try harder` is output and the program skips over the nested `if else`. However, if the `boolean` expression is false (when `GPA` is greater than or equal to 3.5), the false part executes. This second `if else` statement is the false part of the first `if else`. It determines if `GPA` is high enough to qualify for either the dean's list or the president's list.

When implementing multiple selection with `if else` statements, it is important to use proper indentation so the code executes as its written appearance suggests. The readability that comes from good indentation habits saves time during program implementation. To illustrate the flexibility you have in formatting, the previous multiple selection may be implemented in the following preferred manner to line up the three different paths through this control structure:

```
if (GPA < 3.5)
  System.out.println("Try harder");
else if (GPA < 4.0)
  System.out.println("You made the dean's list");
else
  System.out.println("You made the president's list");
```

Another Example — Determining Letter Grades

Some schools use a scale like the following to determine the proper letter grade to assign to a student. The letter grade is based on a percentage representing a weighted average of all of the work for the term. Based on the following table, all percentage values must be in the range of 0.0 through 100.0:

Value of Percentage	Assigned Grade
90.0 ≤ percentage ≤ 100.0	A
80.0 ≤ percentage < 90.0	B
70.0 ≤ percentage < 80.0	C
60.0 ≤ percentage < 70.0	D
0.0 ≤ percentage < 60.0	F

This problem is an example of choosing one action from more than two different actions. A method to determine the range `weightedAverage` falls into could be implemented with unnecessarily long separate `if` statements:

```
public String letterGrade(double weightedAverage) {
  String result = "";
  if(weightedAverage >= 90.0 && weightedAverage <= 100.0)
    result = "A";
  if(weightedAverage >= 80.0 && weightedAverage < 90.0)
    result = "B";
  if(weightedAverage >= 70.0 && weightedAverage < 80.0)
    result = "C";
  if(weightedAverage >= 60.0 && weightedAverage < 70.0)
    result = "D";
  if(weightedAverage >=  0.0 && weightedAverage < 60.0)
    result = "F";
  return result;
}
```

When given the problem of choosing from one of six actions, it is better to use multiple selection, not guarded action. The preferred multiple selection implementation—shown below—is more efficient at runtime. The solution above is correct, but it requires the evaluation of six complex `boolean` expression every time. The solution shown below, with nested `if else` statements,

stops executing when the first boolean test evaluates to true. The true part executes and all of the remaining nested `if else` statements are skipped.

Additionally, the multiple selection pattern shown next is less prone to intent errors. It ensures that an error message will be returned when `weightedAverage` is outside the range of 0.0 through 100.0 inclusive. There is a possibility, for example, an argument will be assigned to `weightedAverage` as 777 instead of 77. Since 777 >= 90.0 is `true`, the method in the code above could improperly return an emtpy String when a `"C"` would have likely been the intended result.

The nested `if else` solution first checks if `weightedAverage` is less than 0.0 or greater than 100.0. In this case, an error message is concatenated instead of a valid letter grade.

```
if ((weightedAverage < 0.0) || (weightedAverage > 100.0))
  result = weightedAverage + " not in the range of 0.0 through 100.0";
```

If `weightedAverage` is out of range—less than 0 or greater than 100—the result is an error message and the program skips over the remainder of the nested `if else` structure. Rather than getting an incorrect letter grade for percentages less than 0 or greater than 100, you get a message that the value is out of range.

However, if the first `boolean` expression is `false`, then the remaining nested `if else` statements check the other five ranges specified in the grading policy. The next test checks if `weightedAverage` represents an A. At this point, `weightedAverage` is certainly less than or equal to 100.0, so any value of `weightedAverage >= 90.0` sets `result` to `"A"`.

```
public String letterGrade(double weightedAverage) {
   String result = "";
   if ((weightedAverage < 0.0) || (weightedAverage > 100.0))
     result = weightedAverage + " not in the range of 0.0 through 100.0";
   else if (weightedAverage >= 90)
     result = "A";
   else if (weightedAverage >= 80.0)
     result = "B";
   else if (weightedAverage >= 70.0)
     result = "C";
   else if (weightedAverage >= 60.0)
     result = "D";
   else
     result = "F";
   return result;
 }
```

The return value depends on the current value of `weightedAverage`. If `weightedAverage` is in the range and is also greater than or equal to 90.0, then “A” will be the `result`. The program skips over all other statements after the first `else`. If `weightedAverage == 50.0`, then all `boolean` expressions are `false` and the program executes the action after the final `else`; `"F"` is concatenated to `result`.

Testing Multiple Selection

Consider how many method calls should be made to test the `letterGrade` method with multiple selection—or for that matter, any method or segment of code containing multiple selection. To test this particular example to ensure that multiple selection is correct for all possible percentage arguments, the method could be called with all numbers in the range from -1.0 through 101.0. However, this would require an infinite number of method calls for arguments such as 1.000000000001 and 1.999999999999, for example. With integers, it would be a lot easier, but still tedious. Such testing is unnecessary.

First consider a set of test data that executes every possible branch through the nested if else. Branch coverage testing means observing what happens when every statement (including the true and false parts) of a nested if else executes once.

Testing should also include the cut-off (boundary) values. This extra effort could go a long way. For example, testing the cut-offs might avoid situations where students with 90.0 are accidentally shown to have a letter grade of B rather than A. This would occur when the Boolean expression `(percentage >= 90.0)` is accidentally coded as `(percentage > 90.0)`. The arguments of 60.0, 70.0, 80.0, and 90.0 complete the boundary testing of the code above.

The best testing strategy is to select test values that combine branch and boundary testing at the same time. For example, a percentage of 90.0 should return `"A"`. The value of 90.0 not only checks the path for returning an A, it also tests the boundary—90.0 as one cut-off. Counting down by tens to 60 checks all boundaries. However, this still misses one path: the one that sets result to `"F"`. Adding 59.9 completes the test driver. These three things are necessary to correctly perform branch coverage testing:

- Establish a set of data that executes all branches (all possible paths through the multiple selection) and boundary (cut-off) values.
- Execute the portion of the program containing the multiple selection for all selected data values. This can be done with a test method and several assertions.
- Observe that the all assertions pass (green bar).

For example, the following data set executes all branches of `letterGrade` while checking the boundaries and the These two methods that follow do both branch and boundary testing.

```
101.1  -0.1  0.0  59.9  60.0  69.9  70.0  79.9  80.0 89.9  90.0  99.9  100.0
```

```
@Test
public void testLetterGradeWhenArgumentNotInRange() {
  assertEquals("100.1 not in the range of 0.0 through 100.0", letterGrade(100.1));
  assertEquals("-0.1 not in the range of 0.0 through 100.0", letterGrade(-0.1));
}

@Test
public void testLetterGradeWhenArgumentIsInRange() {
  assertEquals("F", letterGrade(0.0));
  assertEquals("F", letterGrade(59.9));
  assertEquals("D", letterGrade(60.0));
  assertEquals("D", letterGrade(69.9));
  assertEquals("C", letterGrade(70.0));
  assertEquals("C", letterGrade(79.9));
  assertEquals("B", letterGrade(80.0));
  assertEquals("B", letterGrade(89.9));
  assertEquals("A", letterGrade(90.0));
  assertEquals("A", letterGrade(99.9));
  assertEquals("A", letterGrade(100.0));
}
```

Self-Check

5-4 Which value of `weightedAverage` detects the intent error in the following code when you see this feedback from JUnit org.junit.ComparisonFailure: expected:<[A]> but was:<[B]>?

```
if(weightedAverage > 90)
  result = "A";
else if(weightedAverage >=80)
  result = "B";
else if(weightedAverage >= 70)
  result = "C";
else if(weightedAverage >= 60)
  result = "D";
else
   result = "F";
```

5-5 What string would be incorrectly assigned to `letterGrade` for this argument (answer to 5-4)?

5-6 Would you be happy if your grade were incorrectly computed in this manner?

Use method `currentConditions` *to answer the questions that follow*

```
  public String currentConditions(int currentTemp) {
    String result;
    if (currentTemp <= -40)
      result = "dangerously cold";
    else if (currentTemp <= 0)
      result = "freezing";
    else if (currentTemp <= 10)
      result = "cold";
    else if (currentTemp <= 20)
      result = "mild";
    else if (currentTemp <= 30)
      result = "warm";
    else if (currentTemp <= 40)
      result = "hot";
    else if (currentTemp <= 45)
      result = "very hot";
    else
      result = "dangerously hot";
    return result;
  }
}
```

5-7 List the range of integers that would cause `currentConditions` to return `warm`.

5-8 List a range of integers that would cause `currentConditions` to return `freezing`.

5-9 Establish a list of arguments that tests the boundaries in `currentConditions`.

5-10 Establish a list of arguments that tests the branches in `currentConditions`.

5-11 Write in the correct expected value so each assertion passes.

```
import static org.junit.Assert.*;
import org.junit.Test;

public class LittleWeatherTest {

  @Test
  public void testLittleWeather() {

    assertEquals("_________________", currentConditions(-41));

    assertEquals("_________________", currentConditions(-40));

    assertEquals("_________________", currentConditions(-39));

    assertEquals("_________________", currentConditions(0));

    assertEquals("_________", currentConditions(1));

    assertEquals("_________", currentConditions(10));

    assertEquals("_________", currentConditions(11));

    assertEquals("_________", currentConditions(20));

    assertEquals("_________", currentConditions(21));

    assertEquals("_________", currentConditions(30));

    assertEquals("_________", currentConditions(31));

    assertEquals("_________", currentConditions(40));
```

```
    assertEquals("________", currentConditions(41));
    assertEquals("________", currentConditions(45));
    assertEquals("________", currentConditions(46));
  }
```

Answers to Self-Check Questions

5-1 -a `dubious`
`failing`
-b `dubious`
-c `deleteRecord`

5-2 -a `true`
`after if else` *The last prinltn is not part of the else. It always executes*
-b `zero or pos`
-c `x is low`
-d `neg`

5-3
```
if(option % 2 == 0)
  System.out.println( "Your School" );
else
  System.out.println( "Your name" );
```

5-4 90

5-5 B (instead of the deserved A).

5-6 I wouldn't be happy; I doubt you would either.

5-7 21 through 30 inclusive

5-8 -39 through 0 inclusive

5-9 -40 0 10 20 30 40 45

5-10 any integer < -41, -15 (or any integer in the range of -30 through -1), 5, 15, 25, 35, 42, and any integer > 46

5-11
```
assertEquals("dangerously cold", currentConditions(-41));
assertEquals("dangerously cold", currentConditions(-40));
assertEquals("freezing", currentConditions(-39));
assertEquals("freezing", currentConditions(0));
assertEquals("cold", currentConditions(1));
assertEquals("cold", currentConditions(10));
assertEquals("mild", currentConditions(11));
assertEquals("mild", currentConditions(20));
assertEquals("warm", currentConditions(21));
assertEquals("warm", currentConditions(30));
assertEquals("hot", currentConditions(31));
assertEquals("hot", currentConditions(40));
assertEquals("very hot", currentConditions(41));
assertEquals("very hot", currentConditions(45));
assertEquals("dangerously hot", currentConditions(46));
```

Chapter 6

Repetition

Goals

This chapter introduces the third major control structure—repetition (sequential and selection being the first two). Repetition is discussed within the context of two general algorithmic patterns—the determinate loop and the indeterminate loop. Repetitive control allows for execution of some actions either a specified, predetermined number of times or until some event occurs to terminate the repetition. After studying this chapter, you will be able to

- Use the Determinate Loop pattern to execute a set of statements until an event occurs to stop.
- Use the Indeterminate Loop pattern to execute a set of statements a predetermined number of times
- Design loops

6.1 Repetition

Repetition refers to the repeated execution of a set of statements. Repetition occurs naturally in non-computer algorithms such as these:

- For every name on the attendance roster, call the name. Write a checkmark if present.
- Practice the fundamentals of a sport
- Add the flour ¼-cup at a time, whipping until smooth.

Repetition is also used to express algorithms intended for computer implementation. If something can be done once, it can be done repeatedly. The following examples have computer-based applications:

- Process any number of customers at an automated teller machine (ATM)
- Continuously accept hotel reservations and cancellations
- While there are more fast-food items, sum the price of each item
- Compute the course grade for every student in a class
- Microwave the food until either the timer reaches 0, the cancel button is pressed, or the door opens

Many jobs once performed by hand are now accomplished by computers at a much faster rate. Think of a payroll department that has the job of producing employee paychecks. With only a few employees, this task could certainly be done by hand. However, with several thousand employees, a very large payroll department would be necessary to compute and generate that many paychecks by hand in a timely fashion. Other situations requiring repetition include, but are

certainly not limited to, finding an average, searching through a collection of objects for a particular item, alphabetizing a list of names, and processing all of the data in a file.

The Determinate Loop Pattern

Without the selection control structures of the preceding chapter, computers are little more than nonprogrammable calculators. Selection control makes computers more adaptable to varying situations. However, what makes computers powerful is their ability to repeat the same actions accurately and very quickly. Two algorithmic patterns emerge. The first involves performing some action a specific, predetermined (known in advance) number of times. For example, to find the average of 142 test grades, you would repeat a set of statements exactly 142 times. To pay 89 employees, you would repeat a set of statements 89 times. To produce grade reports for 32,675 students, you would repeat a set of statements 32,675 times. There is a pattern here.

In each of these examples, a program requires that the exact number of repetitions be determined somehow. The number of times the process should be repeated must be established before the loop begins to execute. You shouldn't be off by one. Predetermining the number of repetitions and then executing some appropriate set of statements precisely a predetermined number of times is referred to here as the Determinate Loop pattern.

Algorithmic Pattern: Determinate Loop

Pattern: Determinate Loop

Problem: Do something exactly *n* times, where *n* is known in advance.

Outline: Determine *n* as the number of times to repeat the actions
  Set a counter to 1
  While counter <= *n*, do the following
    Execute the actions to be repeated

Code Example:

```
// Print the integers from 1 through n inclusive
int counter = 1;
int n = 5;
while (counter <= n) {
  System.out.println(counter);
  counter = counter + 1;
}
```

The Java `while` statement can be used when a determinate loop is needed.

***General Form:* `while` statement**

`while (`*loop-test***`) {`**
  repeated-part
`}`

Example

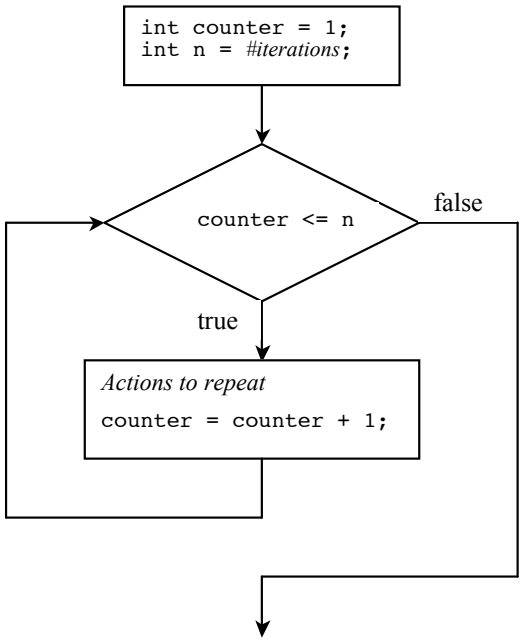
```
int start = 1;
int end = 6;
while (start < end) {
  System.out.println(start + " " + end);
  start = start + 1;
  end = end - 1;
}
```

Output

```
1 6
2 5
3 4
```

The *loop-test* is a `boolean` expression that evaluates to either true or false. The *repeated-part* may be any Java statement, but it is usually a set of statements enclosed in `{` and `}`.
When a `while` loop is encountered, the loop test evaluates to either true or false. If true, the repeated part executes. This process continues while (as long as) the loop test is true.

Flow Chart View of one Indeterminate Loop



To implement the Determinate Loop Pattern you can use some `int` variable—named `n` here—to represent how often the actions must repeat. However, other appropriate variable names are certainly allowed, such as `numberOfEmployees`. The first thing to do is determine the number of repetitions somehow. Let `n` represent the number of repetitions.

n = number of repetitions

The number of repetitions may be input, as in `int n = keyboard.nextInt();` or `n` may be established at compiletime, as in `int n = 124;` or `n` may be passed as an argument to a method as shown in the following method heading.

```
// Return the sum of the first n integers.
// Precondition: n >= 0
public int sumOfNInts(int n)
```

The method call `sumOfNInts(4)` should return the sum of all positive integers from 1 through 4 inclusive or 1 + 2 + 3 +4 = 10. The following test method shows four other expected values with different values for n.

```
@Test
public void testSumOfNInts() {
  assertEquals(0, sumOfNInts(0));
  assertEquals(1, sumOfNInts(1));
  assertEquals(3, sumOfNInts(2));
  assertEquals(1 + 2 + 3 + 4 + 5 + 6 + 7, sumOfNInts(7));
}
```

Once `n` is known, another `int` variable, named `counter` in the `sumOfNInts` method below, helps control the number of loop iterations.

```
// Return the sum of the first n integers
public int sumOfNInts(int n) {
  int result = 0;

  int counter = 1;
  // Add counter to result as it changes from 1 through n
  while (counter <= n) {
    result = result + counter;
    counter = counter + 1;
  }
  return result;
}
```

The action to be repeated is incrementing result by the value of counter as it progresses from 1 through n. Incrementing `counter` at each loop iteration gets the loop one step closer to termination.

Determinate Loop with Strings

Sometimes an object carries information to determine the number of iterations to accomplish the task. Such is the case with `String` objects. Consider `numSpaces(String)` that returns the number of spaces in the String argument. The following assertions must pass

```
@Test
public void testNumSpaces() {
  assertEquals(0, numSpaces(""));
  assertEquals(2, numSpaces(" a "));
  assertEquals(7, numSpaces("  a  bc   "));
  assertEquals(0, numSpaces("abc"));
}
```

The solution employs the determinate lop pattern to look at each and every character in the String. In this case, `str.length()` represents the number of loop iterations. However, since the characters in a string are indexed from 0 through its length() – 1, index begins at 0.

```
// Return the number of spaces found in str.
public int numSpaces(String str) {
  int result = 0;
  int index = 0;
  while (index < str.length()) {
    if (str.charAt(index) == ' ')
      result = result +1;
    index++;
  }
  return result;
}
```

Infinite Loops

It is possible that a loop may never execute, not even once. It is also possible that a `while` loop never terminates. Consider the following `while` loop that potentially continues to execute until external forces are applied such as terminating the program, turning off the computer or having a power outage. This is an infinite loop, something that is usually undesirable.

```
// Print the integers from 1 through n inclusive
int counter = 1;
int n = 5;
while (counter <= n) {
  System.out.println(counter);
}
```

The loop repeats virtually forever. The termination condition can never be reached. The loop test is always true because there is no statement in the repeated part that brings the loop closer to the termination condition. It should increment counter so it eventually becomes greater than to make the loop test is false. When writing `while` loops, make sure the loop test eventually becomes false.

Self-Check

6-1 Write the output from the following Java program fragments:

```
int n = 3;
int counter = 1;
while (counter <= n) {
  System.out.print(counter + "  ");
  counter = counter + 1;
}
```

```
int low = 1;
int high = 9;
while (low < high) {
  System.out.println(low + " " + high);
  low = low + 1;
  high = high - 1;
}
```

```
int last = 10;
int j = 2;
while (j <= last) {
  System.out.print(j + "  ");
  j = j + 2;
}
```

```
int counter = 10;
// Tricky, but an easy-to-make mistake
while (counter >= 0); {
  System.out.println(counter);
  counter = counter - 1;
}
```

6-2 Write the number of times “Hello” is printed. “Zero” and “Infinite” are valid answers.

```
int counter = 1;
int n = 20;
while (counter <= n) {
  System.out.print("Hello ");
  counter = counter + 1;
}
```

```
int j = 1;
int n = 5;
while (j <= n) {
  System.out.print("Hello ");
  n = n + 1;
  j = j + 1;
}
```

```
int counter = 1;
int n = 5;
while (counter <= n) {
  System.out.print("Hello ");
  counter = counter + 1;
}
```

```
// Tricky
int n = 5;
int j = 1;
while (j <= n)
  System.out.print("Hello ");
  j = j + 1;
```

6-3 Implement method `factorial` that return n!. `factorial(0)` must return 1, `factorial(1)` must return 1, `factorial(2)` must return 2*1, `factorial(3)` must return 3*2*1, and `factorial(4)` must return is 4*3*2*1. The following assertions must pass.

```
@Test
public void testFactorial() {
  assertEquals(1, factorial(0));
  assertEquals(1, factorial(1));
  assertEquals(2, factorial(2));
  assertEquals(6, factorial(3));
  assertEquals(7 * 6 * 5 * 4 * 3 * 2 * 1, factorial(7));
}
```

6-4 Implement method `duplicate` that returns a string where every letter is duplicated. Hint: Create an empty String referenced by `result` and concatenate each character in the argument to result twice. The following assertions must pass.

```
@Test
public void testDuplicate() {
  assertEquals("", duplicate(""));
  assertEquals("  ", duplicate(" "));
  assertEquals("zz", duplicate("z"));
  assertEquals("xxYYzz", duplicate("xYz"));
  assertEquals("1122334455", duplicate("12345"));
}
```

6.2 Indeterminate Loop Pattern

It is often necessary to execute a set of statements an undetermined number of times. For example, to process report cards for *every* student in a school where the number of students changes from semester to semester. Programs cannot always depend on prior knowledge to determine the exact number of repetitions. It is often more convenient to think in terms of "process a report card for all students" rather than "process precisely 310 report cards." This leads to a recurring pattern in algorithm design that captures the essence of repeating a process an unknown number of times. It is a pattern to help design a process of iterating until something occurs to indicate that the looping is finished. The **Indeterminate Loop pattern** occurs when the number of repetitions is not known in advance.

Algorithmic Pattern

Pattern: Indeterminate Loop

Problem: A process must repeat an unknown number of times.

Outline:
```
while (the termination condition has not occurred) {
    perform the actions
    do something to bring the loop closer to termination
}
```

Code Example
```
// Return the greatest common divisor of two positive integers.
public int GCD(int a, int b) {
  while (b != 0) {
    if (a > b)
      a = a - b;
    else
      b = b - a;
  }
  return a;
}
```

The code example above is an indeterminate loop because the algorithm cannot determine how many times a must be subtracted from b or b from a. The loop repeats until there is nothing more to subtract. When b becomes 0, the loop terminates. When the following test method executes, the loop iterates a varying number of times:

```
@Test
public void testGCD() {
  assertEquals(2, GCD(6, 4));
  assertEquals(7, GCD(7, 7));
  assertEquals(3, GCD(24, 81));
  assertEquals(5, GCD(15, 25));
}
```

GCD(6, 4) → 2

a	b
6	4
2	4
2	2
2	0

GCD(7, 7) → 7

a	b
7	7
7	0

GCD(24, 81) → 3

a	b
24	81
24	57
24	33
24	9
15	9
6	9
6	3
3	3
3	0

GCD(15, 25) → 5

a	b
15	25
15	10
5	10
5	5
5	0

The number of iterations in the four assertions ranges from 1 to 8. However, GCD(1071, 532492) results in 285 loop iterations to find there is no common divisor other than 1. The following alternate algorithm for GCD(a, b) using modulus arithmetic more quickly finds the GCD in seven iterations because b approaches 0 more quickly with %.

```
// Return the greatest common divisor of two
// positive integers with fewer loop iterations
public int GCD(int a, int b) {
  while (b != 0) {
    int temp = a;
    a = b;
    b = temp % b;
  }
  return a;
}
```

a	b
532492	1071
1071	205
205	46
46	21
21	4
4	1
1	0

Indeterminate Loop with Scanner(String)

Sometimes a stream of input from the keyboard or a file needs to be read until there is no more needed input. The amount of input may not be known until there is no more. A convenient way to expose this processing is to use a Scanner with a String argument to represent input from the keyboard or a file.

```
// Constructs a new Scanner that produces values scanned from the specified
// string. The parameter source is the string to scan
public void Scanner(String source)
```

Scanner has convenient methods to determine if there is any more input of a certain type and to get the next value of that type. For example to read white space separated strings, use these two methods from `java.util.Scanner`.

```
// Returns true if this scanner has another token in its input.
// This method may block while waiting for keyboard input to scan.
public boolean hasNext()

// Return the next complete token as a string.
public String next()
```

The following test methods demonstrates how `hasNext()` will eventually return false after `next()` has been called for every token in scanner's string.

```
@Test
public void showScannerWithAStringOfStringTokens() {
  Scanner scanner = new Scanner("Input with four tokens");
  assertTrue(scanner.hasNext());
  assertEquals("Input", scanner.next());
  assertTrue(scanner.hasNext());
  assertEquals("with", scanner.next());
  assertTrue(scanner.hasNext());
  assertEquals("four", scanner.next());
  assertTrue(scanner.hasNext());
  assertEquals("tokens", scanner.next());

  // Scanner has scaned all tokens, so hasNext() should now be false.
  assertFalse(scanner.hasNext());
}
```

You can also have the `String` argument in the `Scanner` constructor contain numeric data. You have used `nextInt()` before in Chapter 2's console based programs.

```
// Returns true if the next token in this scanner's input
// can be interpreted as an int value.
public boolean hasNextInt()

// Scans the next token of the input as an int.
public int nextInt()
```

The following test method has an indeterminate loop that repeats as long as there is another valid integer to read.

```
@Test
public void showScannerWithAStringOfIntegerTokens() {
  Scanner scanner = new Scanner("80 70 90");
  // Sum all integers found as tokens in scanner
  int sum = 0;
  while (scanner.hasNextInt()) {
    sum = sum + scanner.nextInt();
  }
  assertEquals(240, sum);
}
```

`Scanner` also has many such methods whose names indicate what they do: `hasNextDouble()` with `nextDouble()`, `hasNextLine()` with `nextLine()`, and `hasNextBoolean()` with `nextBoolean()`.

A Sentinel Loop

A **sentinel** is a specific input value used only to terminate an indeterminate loop. A sentinel value should be the same type of data as the other input. However, this sentinel must not be treated the same as other input. For example, the following set of inputs hints that the input of -1 is the event that terminates the loop and that -1 is not to be counted as a valid test score. If it were counted as a test score, the average would not be 80.

Dialogue

```
Enter test score #1 or -1.0 to quit: 80
Enter test score #2 or -1.0 to quit: 90
Enter test score #3 or -1.0 to quit: 70
Enter test score #4 or -1.0 to quit: -1
Average of 3 tests = 80.0
```

This dialogue asks the user either to enter test scores or to enter -1.0 to signal the end of the data. With **sentinel loops**, a message is displayed to inform the user how to end the input. In the dialogue above, -1 is the sentinel. It could have some other value outside the valid range of inputs, any negative number, for example.

Since the code does not know how many inputs the user will enter, an indeterminate loop should be used. Assuming that the variable to store the user input is named `currentInput`, the termination condition is `currentInput == -1`. The loop should terminate when the user enters a value that flags the end of the data. The loop test can be derived by taking the logical negation of the termination condition. The `while` loop test becomes `currentInput != -1`.

```
while (currentInput != -1)
```

The value for `currentInput` must be read before the loop. This is called a "priming read," which goes into the first iteration of the loop. Once inside the loop, the first thing that is done is to process the `currentInput` from the priming read (add its value to `sum` and add `1` to `n`). Once that is done, the second `currentInput` is read at the "bottom" of the loop. The loop test evaluates next. If `currentInput != -1`, the second input is processed. This loop continues until the user enters -1. Immediately after the `nextInt` message at the bottom of the loop, `currentValue` is compared to `SENTINEL`. When they are equal, the loop terminates. The `SENTINEL` is not added to the running sum, nor is 1 added to the count. The awkward part of this algorithm is that the loop is processing data read in the *previous* iteration of the loop.

The following method averages any number of inputs. It is an instance of the Indeterminate Loop pattern because the code does not assume how many inputs there will be.

```
import java.util.Scanner;
// Find an average by using a sentinel of -1 to terminate the loop
// that counts the number of inputs and accumulates those inputs.
public class DemonstrateIndeterminateLoop {

  public static void main(String[] args) {
    double accumulator = 0.0; // Maintain running sum of inputs
    int n = 0;                // Maintain total number of inputs
    double currentInput;
    Scanner keyboard = new Scanner(System.in);

    System.out.println("Compute average of numbers read.");
    System.out.println();
    System.out.print("Enter number or -1 to quit: ");
    currentInput = keyboard.nextDouble();

    while (currentInput != -1) {
      accumulator = accumulator + currentInput; // Update accumulator
      n = n + 1; // Update number of inputs so far
      System.out.print("Enter number or -1 to quit: ");
      currentInput = keyboard.nextDouble();
    }

    if (n == 0)
      System.out.println("Can't average zero numbers");
    else
      System.out.println("Average: " + accumulator / n);
  }
}
```

Dialogue

```
Compute average of numbers read.

Enter number or -1.0 to quit: 70.0
Enter number or -1.0 to quit: 90.0
```

```
Enter number or -1.0 to quit: 80.0
Enter number or -1.0 to quit: -1.0
Average: 80.0
```

The following table traces the changing state of the important variables to simulate execution of the previous program. The variable named `accumulator` maintains the running sum of the test scores. The loop also increments `n` by +1 for each valid `currentInput` entered by the user. Notice that `-1` is not treated as a valid `currentInput`.

Iteration Number	`currentInput`	`accumulator`	`n`	`currentInput != SENTINEL`
Before the loop	NA	0.0	0	NA
Loop 1	70.0	70.0	1	True
Loop 2	90.0	160.0	2	True
Loop 3	80.0	240.0	3	True
After the loop	NA	240.0	3	NA

Self-Check

6-5 Determine the value assigned to average for each of the following code fragments by simulating execution when the user inputs 70.0, 60.0, 80.0, and -1.0.

```
Scanner keyboard = new Scanner(System.in);
int n = 0;
double accumulator = 0.0;
double currentInput = keyboard.nextDouble();
while (currentInput != -1.0) {
  currentInput = keyboard.nextDouble();
  accumulator = accumulator  + currentInput; // Update accumulator
  n = n + 1; // Update total # of inputs
}
double average = accumulator / n;
```

6-6 If you answered 70.0 for 6-5, try again until you get an answers for !- 70.

6-7 What is the value of `numberOfWords` after this code executes with the dialogue shown (read the input carefully).

```
String SENTINEL = "QUIT";
Scanner keyboard = new Scanner(System.in);
String theWord = "";
int numberOfWords = 0;
System.out.println("Enter words or 'QUIT' to quit");
while (!theWord.equals(SENTINEL)) {
  numberOfWords = numberOfWords + 1;
  theWord = keyboard.next();
}
System.out.println("You entered " + numberOfWords + " words.");
```

Output

```
Enter words or 'QUIT' to quit
The quick brown fox quit and then jumped over the lazy dog. QUIT
You entered ___ words.
```

Reading Input from a Text File

In programs that require little data, interactive input suffices. However, some problems involve large amounts of data. The input would have to be typed in from the keyboard many times during implementation and testing. That much interactive input would be tedious and error-prone. So here we will be read the data from an external file instead.

Consider counting the number of words in a book for example, specifically "Tale of Two Cities" by Charles Dickens that begins like this:

```
Book the First--Recalled to Life


I. The Period


It was the best of times,
it was the worst of times,
it was the age of wisdom,
it was the age of foolishness,
it was the epoch of belief,
it was the epoch of incredulity,
it was the season of Light,
it was the season of Darkness,
it was the spring of hope,
it was the winter of despair,
we had everything before us,
we had nothing before us,
we were all going direct to Heaven,
we were all going direct the other way--
in short, the period was so far like the
present period, that some of its noisiest
authorities insisted on its being received,
for good or for evil, in the superlative
degree of comparison only.
```

With some additional code, we can use the familiar Scanner class to read until the end of the book when stored in a file. However, this time a new `File` object will be needed. And this requires some understanding of exception handling.

Exception Handling when a File is Not Found

When programs run, errors occur. Perhaps an arithmetic expression results in division by zero, or an array subscript is out of bounds, or there is an attempt to read a file from a disk using a specific file name that does not exist. Or perhaps, the expression in an array subscript is negative or 1 greater than the capacity of that array. Programmers have at least two options for dealing with these types of exception:

- Ignore the exception and let the program terminate
- Handle the exception

However, in order to read from an input file, you cannot ignore the exception. Java forces you to try to handle the exceptional event. Here is the code the tries to have a Scanner object read from an input file named taleOf2.txt. Notice the argument is now a new File object.

```
Scanner inFile = new Scanner(new File("taleOf2.txt"));
```

This will not compile. Because it isd possible the file `"taleOf2.txt"` may not be found at runtime, the code may throw a `FileNotFoundException`. In this type of exception (called a checked exception), Java requires that you put the construction of a new `File` object in a `try` block—the keyword `try` followed by the code wrapped in a block, `{ }`.

```
try {
    code that may throw an exception when an exception is thrown
}
catch (Exception anException) {
    code that executes only if an exception is thrown from code in the above try block.
}
```

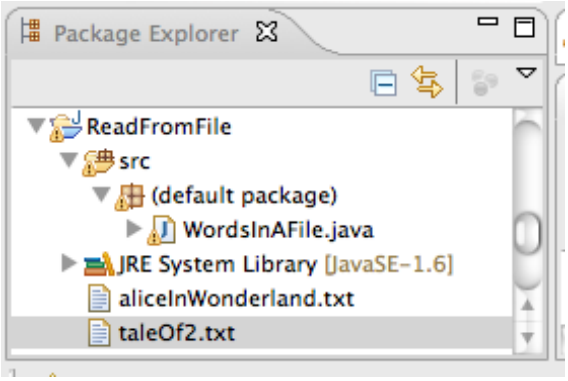
Every `try` block must be followed by a at least one `catch` block—the keyword `catch` followed by the anticipated exception as a parameter and code wrapped in a block. The `catch` block contains the code that executes when the code in the `try` block causes an exception to be thrown (or called a method that throws an exception). So to get a Scanner object to try to read from an input file, you need code like this.

```
String fileName = "taleOf2.txt";
Scanner inFile = null;
try {
  inFile = new Scanner(new File(fileName));
}
catch (FileNotFoundException fnfe) {
  System.out.println("The file '" + fileName + " was not found");
}
```

When this code tries to create a new File object to be read from, one of two things happens:

1. `inFile` successfully refers to a `Scanner` object that uses the text in the found file as input. In this case, the catch block (with the `println`) is skipped and we can send `hasNext` and `next` messages to `inFile`.
2. The file is not found and the catch block executes. In this case, `inFile` will remain null and we can not send `hasNext` message to `inFile`.

At this point, a program could read input from the file word by word as long as the file was found. Most easily, the input file must be in the same directory as the program reading the file (the file could be in a different folder or on a different disk drive as long as the file path is specified). However, if you are using Eclipse, the file must be added to the project rather than the default package of the src directory. Your Eclipse project files might be arranged to something like this.



The file containing input to be read could be created as a file in any text editor or copied into the correct folder.

The following program that computes the number of words found needs a few additional input for the two new types being used: File and FileNotFoundException.

```
// This program reads one particular file word by word to
// get a rough count of the number of words in that file.

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class WordsInAFile {

  public static void main(String[] args) {
    String fileName = "taleOf2.txt";
    Scanner inFile = null;
    try {
      inFile = new Scanner(new File(fileName));
    } catch (FileNotFoundException fnfe) {
      System.out.println("The file '" + fileName + " was not found");
    }

    if(inFile != null) {

      int numberOfWords = 0;
      while (inFile.hasNext()) {
        String aWord = inFile.next();
        numberOfWords = numberOfWords + 1;
      }
      System.out.println(fileName + " has " + numberOfWords + " words.");
    }
  }

}
```

Output

```
taleOf2.txt has 135475 words.
```

The **for** Statement

Java has several structures for implementing repetition. The `while` statement shown above can be used to implement indeterminate and determinate loop patterns. Java also has added a for loop that combines all looping logic into more compact code. The `for` loop was added to programming languages because the Determinate Loop Pattern arises so often. Here is the general form of the Java `for` loop:

***General Form:* `for` statement**

`for(`*initial-statement***`;`** *loop-test***`;`** *update-step***`)`** **`{`**
    *repeated-part***`;`**
`}`

The following `for` statement shows the three components that maintain the Determinate Loop pattern: the initialization (`n = 5` and `j = 1`), the loop test for determining when to stop (`j <= n`), and the update step (`j = j + 1`) that brings the loop one step closer to terminating.

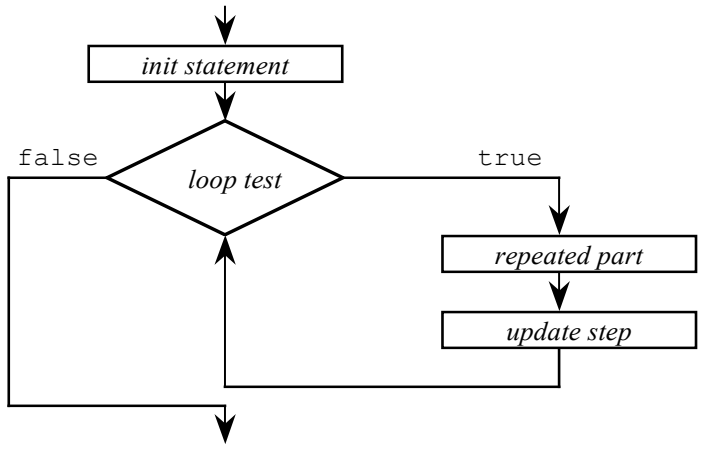
```
// Predetermined number of iterations
int n = 5;
for (int j = 1; j <= n; j = j + 1) {
  // Execute this block n times
}
```

In the preceding `for` loop, `j` is first assigned the value of 1. Next, `j <= n` (`1 <= 5`) evaluates to `true` and the block executes. When the statements inside the block are done, `j` increments by 1 (`j=j+1`). These three components ensure that the block executes precisely `n` times.

```
j = 1       // Initialize counter
j <= n      // Loop test
j = j + 1   // Update counter
```

When a `for` loop is encountered, the *initial-statement* is executed first and only once. The *loop-test* evaluates to either `true` or `false` before each execution of the *repeated-part*. The *update-step* executes after each iteration of the repeated part. This process continues until the loop test evaluates to `false`.

Flowchart view of a for loop



The following `for` statement simply displays the value of the loop counter named `j` as it ranges from 1 through 5 inclusive:

```
int n = 5;
for (int j = 1; j <= n; j = j + 1) {
  System.out.print(j + " ");
}
```

Output

```
1 2 3 4 5
```

Other Increment and Assignment Operators

Assignment operations alter computer memory even when the variable on the left of `=` is also involved in the expression to the right of `=`. For example, the variable `int j` is incremented by `1` with this assignment operation:

```
j = j + 1;
```

This type of update—incrementing a variable—is performed so frequently that Java offers operators with the express purpose of incrementing variables. The `++` and `--` operators **increment** and **decrement** a variable by 1, respectively. For example, the expression `j++` adds `1` to the value of `j`, and the expression `x--` reduces `x` by 1. The `++` and `--` unary operators alter the numeric variable that they follow (see the table below).

Statement	Value of `j`
`int j = 0;`	0
`j++;`	1
`j++;`	2
`j--;`	1

So, within the context of the determinate loop, the update step can be written as `j++` rather than `j = j + 1`. This `for` loop

```
for (int j = 1; j <= n; j = j + 1) {
  // ...
}
```

may also be written with the `++` operator for equivalent behavior:

```
for(int j = 1; j <= n; j++) {
  // ...
}
```

These new assignment operators are shown because they provide a convenient way to increment and decrement a counter in `for` loops. Also, most Java programmers use the `++` operator in `for` loops. You will see them often.

Java has several assignment operators in addition to `=`. Two of them, `+=` and `-=`, add and subtract value from the variable to the left, respectively.

Operator	Equivalent Meaning
`+=`	Increment variable on left by value on right.
`-=`	Decrement variable on left by value on right.

These two new operators alter the numeric variable that they follow.

Statement	Value of j
`int j = 0;`	0
`j += 3;`	3
`j += 4;`	7
`j -= 2;`	5

Whereas the operators **++** and **--** increment and decrement the variable by one, the operators **+=** and **-=** increment and decrement the variable by any amount. The **+=** operator is most often used to accumulate values inside a loop.

The following comparisons show the for loop was designed to put the initialization and the update step together with the loop test. The for loops also use the shorter ++ operator. This makes the code a bit more compact and a bit more difficult to read. However, you will get used to it, especially when the `for` loop will be used extensively in the next chapters.

While loop	For loop equivalent

`public int sumOfNInts(int n) {`
`  int result = 0;`
`  int counter = 1;`
`  while (counter <= n) {`
`  result = result + counter;`
`  counter++;`
`  }`
`  return result;`
`}`
	`public int sumOfNInts(int n) {`
	`  int result = 0;`
	
	`  for (int counter = 1; counter <= n; counter++) {`
	`  result = result + counter;`
	`  }`
	
	`  return result;`
	`}`

`public int numSpaces(String str) {`
`  int result = 0;`
 	
`  int index = 0;`
`  while (index < str.length() ) {`
`  if (str.charAt(index) == ' ')`
`  result++;`
`  index++;`
`  }`
`  return result;`
`}`
	`public int numSpaces(String str) {`
	`  int result = 0;`
	
	`  for (int index = 0; index < str.length();index++) {`
	`  if (str.charAt(index) == ' ')`
	`  result++;`
	`  }`
	
	`  return result;`
	`}`

Self-Check

6-8 Does a `for` loop execute the update step at the beginning of each iteration?

6-9 Must an update step increment the loop counter by `+1`?

6-10 Do `for` loops always execute the repeated part at least once?

6-11 Write the output generated by the following `for` loops.

```
for(int j = 0; j < 5; j++) {
  System.out.print(j + " ");
}

int n = 5;
for( int j = 1; j <= n; j++ ) {
  System.out.print(j + " ");
}

int n = 3;
for (int j = -3; j <= n; j += 2) {
  System.out.print(j + " ");
}
```

```
for( int j = 1; j < 10; j += 2) {
  System.out.print(j + " ");
}

int n = 0;
System.out.print("before ");
for(int j = 1; j <= n; j++) {
  System.out.print( j + " " );
}
System.out.print(" after");

for (int j = 5; j >= 1; j--) {
  System.out.print(j + " ");
}
```

6-12 Write a `for` loop that displays all of the integers from 1 to 100 inclusive on separate lines.

6-13 Write a `for` loop that displays all of the integers from 10 down to 1 inclusive on separate lines.

6.3 Loop Selection and Design

For some people, loops are easy to implement, even at first. For others, infinite loops, being off by one iteration, and intent errors are more common. In either case, the following outline is offered to help you choose and design loops in a variety of situations:

1. Determine which type of loop to use.
2. Determine the loop test.
3. Write the statements to be repeated.
4. Bring the loop one step closer to termination.
5. Initialize variables if necessary.

Determine Which Type of Loop to Use

If the number of repetitions is known in advance or is read as input, it is appropriate to use the Determinate Loop pattern. The `for` statement was specifically designed for this pattern. Although you can use the `while` loop to implement the Determinate Loop pattern, consider using the `for` loop instead. The `while` implementation allows you to omit one of the key parts with no compile time errors thus making any intent errors more difficult to detect and correct. If you leave off one of the parts from a `for` loop, you get an easier-to-detect-and-correct compiletime error.

The Indeterminate Loop pattern is more appropriate when you need to wait until some event occurs during execution of the loop. In this case, use the `while` loop. If you need to process all the data in an input file, consider using a `Scanner` object with one of the `hasNext` methods as the loop test. This is an indeterminate loop.

Determining the Loop Test

If the loop test is not obvious, try writing the conditions that must be true for the loop to terminate. For example, if you want the user to enter `QUIT` to stop entering input, the termination condition is

```
inputName.equals("QUIT")  // Termination condition
```

The logical negation `!inputName.equals("QUIT")` can be used directly as the loop test of a `while` loop.

```
while(! inputName.equals("QUIT")) {
  // . . .
}
```

Write the Statements to Be Repeated

This is why the loop is being written in the first place. Some common tasks include keeping a running sum, keeping track of a high or low value, and counting the number of occurrences of some value. Other tasks that will be seen later include searching for a name in a list and repeatedly comparing all string elements of a list in order to alphabetize it.

Bring the Loop One Step Closer to Termination

To avoid an infinite loop, at least one action in the loop must bring it closer to termination. In a determinate loop this might mean incrementing or decrementing a counter by some specific value. Inputting a value is a way to bring indeterminate loops closer to termination. This happens when a user inputs data until a sentinel is read, for example. In a `for` loop, the repeated statement should be designed to bring the loop closer to termination, usually by incrementing the counter.

Initialize Variables if Necessary

Check to see if any variables used in either the body of the loop or the loop test need to be initialized. Doing this usually ensures that the variables of the loop and the variables used in the iterative part have been initialized. This code attempts to use many variables in expressions before they have been initialized. In certain other languages, these variables are given garbage values and the result is unpredictable. Fortunately, the Java compiler flags these uninitialized variables as errors.

Self-Check

6-14 Which kind of loop best accomplishes these tasks?

- **-a** Sum the first five integers (1 + 2 + 3 + 4 + 5).
- **-b** Find the average for a list of numbers when the size of the list is known.
- **-c** Find the average value for a list of numbers when the size of the list is not known in advance.
- **-d** Obtain a character from the user that must be an uppercase S or Q.

6-15 To design a loop that processes inputs called `value` until `-1` is entered,

- **-a** describe the termination condition.
- **-b** write the Boolean expression that expresses the logical negation of the termination condition. This will be the loop test.

6-16 To design a loop that visits all the characters of `theString`, from the first to the last.

- **-a** describe the termination condition.
- **-b** write the Boolean expression that expresses the logical negation of the termination condition. This will be the loop test.

6-17 Which variables are not initialized but should be?

- **-a** `while(j <= n) {  }`
- **-b** `for(int j = 1; j <= n; j = j + inc) {  }`

Answers to Self Checks

6-1 1 2 3

1 9
2 8
3 7
4 6

2 4 6 8 10

No output, this is an infinite loop, it does nothing. The code between) and ; (an empty statement) until the program is externally terminated.

6-2 20

Infinite since n grows as fast as j, j will always be less than n

5

Infinite since j++ is not part of the loop. Add { and }

6-3
```
public int factorial(int n) {
  int result = 1;
  int counter = 1;
  while (counter <= n) {
    result = result * counter;
    counter++;
  }
  return result;
}
```

6-4
```
public String duplicate(String str) {
  String result = "";
  int index = 0;
  while (index < str.length()) {
    result = result + str.charAt(index) + str.charAt(index);
    index++;
  }
  return result;
}
```

6-5 46.3

6-6 Trace your code again if necessary.

6-7 The answer of 13 includes QUIT. The solution does not include the priming read.

```
You entered 13 words.
```

6-8 No, the update step happens at the end of the loop iteration. The `init` statement happens first, and only once.

6-9 No, you can use increments of any amount, including negative increments (decrements).

6-10 No, consider `for( int j = 1; j < n; j++ ) { /*do nothing*/ }` when `n == 0`.

6-11
```
 0 1 2 3 4
 1 2 3 4 5
-3 -1 1 3
```

```
1 3 5 7 9
before  after
5 4 3 2 1
```

6-12
```
for(int j = 1; j <= 100; j++) {
  System.out.println( j );
}
```

6-13
```
for(int k = 10; k >= 1; k--) {
  System.out.println(k);
}
```

6-14 -a A `for` loop, since number of repetition is known.
-b A `for` loop, since the number of repetitions would be known in advance.
-c An indeterminate loop, perhaps a `while` loop that terminates when the sentinel is read.
-d An indeterminate loop, perhaps a `while` loop that terminates when the sentinel is read.

6-15 -a The `value` just input equals `-1`
-c `value != -1`

6-16 -a An index starting at 0 becomes the length of the string
-c `index < theString.length()`

6-17 -a Both `j` and `n`
-b Both `n` and `inc`

Chapter 7

Arrays

Goals

This chapter introduces the Java array for storing collections of many objects. Individual elements are referenced with the Java subscript operator []. After studying this chapter you will be able to

- declare and use arrays that can store reference or primitive values
- implement methods that perform array processing

7.1 The Java Array Object

Java **array** objects store collections of elements. They allow a large number of elements to be conveniently maintained together under the same name. The first element is at index 0 and the second is at index 1. Array elements may be any one of the primitive types, such as `int` or `double`. Array elements can also be references to any object.

The following code declares three different arrays named `balance`, `id`, and `tinyBank`. It also initializes all five elements of those three arrays. The subscript operator `[]` provides access to individual array elements.

```
// Declare two arrays that can store up to five elements each
double[] balance = new double[5];
String[] id = new String[5];

// Initialize the array of double values
balance[0] = 0.00;
balance[1] = 111.11;
balance[2] = 222.22;
balance[3] = 333.33;
balance[4] = 444.44;

// Initialize all elements in an array of references to String objects
id[0] = "Bailey";
id[1] = "Dylan";
id[2] = "Hayden";
id[3] = "Madison";
id[4] = "Shannon";
```

The values referenced by the arrays can be drawn like this, indicating that the arrays `balance`, and `id`, store collections. `balance` is a collection of primitive values; `id` is a collection of references to String objects.

balance[0]	0.0
balance[1]	1.11
balance[2]	2.22
balance[3]	3.33
balance[4]	4.44

id[0]	"Bailey"
id[1]	"Dylan"
id[2]	"Hayden"
id[3]	"Madison"
id[4]	"Shannon"

The two arrays above were constructed using the following general forms:

***General Form:* Constructing array objects**

type **[ ]** *array-name* **= new** *type* **[***capacity***];**

class-name **[ ]** *array-name* **= new** *class-name* **[***capacity***];**

- *type* specifies the type (either a primitive or reference type) of element that will be stored in the array.
- *array-name* is any valid Java identifier. With subscripts, the array name can refer to any and all elements in the array.
- *capacity* is an integer expression representing the maximum number of elements that can be stored in the array. The capacity is always available through a variable named `length` that is referenced as *array-name*`.length`.

***Example:* array declarations**

```
int[] test = new int[100];            // Store up to 100 integers
double[] number = new double[10000];  // Store up to 10000 numbers
String[] name = new String[500];      // Store up to 500 strings
BankAccount[] customer = new BankAccount[1000]; // 1000 BankAccount references
```

Accessing Individual Elements

Arrays support random access. The individual array elements can be found through subscript notation. A subscript is an integer value between `[` and `]` that represents the index of the element you want to get to. The special symbols `[` and `]` represent the mathematical subscript notation. So instead of x_0, x_1, and x_{n-1}, Java uses `x[0]`, `x[1]`, and `x[n-1]`.

***General Form:* Accessing one array element**

array-name **[***index***]** `// Index should range from 0 to capacity - 1`

The subscript range of a Java array is an integer value in the range of 0 through its capacity - 1. Consider the following array named `x`.

```
double[] x = new double[8];
```

The individual elements of `x` may be referenced using the indexes 0, 1, 2, … 7. If you used `-1` or `8` as an index, you would get an **`ArrayIndexOutOfBoundsException`**. This code assigns values to the first two array elements:

```
// Assign new values to the first two elements of the array named x:
x[0] = 2.6;
x[1] = 5.7;
```

Java uses zero-based indexing. This means that the first array element is accessed with index 0; the same indexing scheme used with `String`. The index 0 means the first element in the

collection. With arrays, the first element is found in subscript notation as `x[0]`. The fifth element is accessed with index 4 or with subscript notation as `x[4]`. This subscript notation allows individual array elements to be displayed, used in expressions, and modified with assignment and input operations. In fact, you can do anything to an individual array element that can be done to a variable of the same type. The array is simply a way to package together a collection of values and treat them as one.

The familiar assignment rules apply to array elements. For example, a `String` literal cannot be assigned to an array element that was declared to store `double` values.

```
// ERROR: x stores numbers, not strings
x[2] = "Wrong type of literal";
```

Since any two `double` values can use the arithmetic operators, numeric array elements can also be used in arithmetic expressions like this:

```
x[2] = x[0] + x[1]; // Store 8.3 into the third array element
```

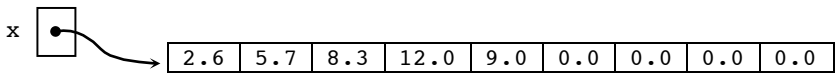
Each array element is a variable of the type declared. Therefore, these two integers will be promoted to double before assignment.

```
x[3] = 12; // Stores 12.0
x[4] = 9;
```

Arrays of primitive `double` values are initialized to a default value of 0.0 (an array of `int`s have elements initialized to 0, arrays of objects to null). The array x originally had all 8 elements to 0.0. After the five assignments above, the array would look like this.

Element Reference	Value
x[0]	2.6
x[1]	5.7
x[2]	8.3
x[3]	12.0
x[4]	9.0
x[5]	0.0
x[6]	0.0
x[7]	0.0

The value of an array is a reference to memory where elements are stored in a contiguous (next to each other) fashion. Here is another view of an array reference value and the elements as the data may exist in the computer's memory.



Out-of-Range Indexes

Java checks array indexes to ensure that they are within the proper range of 0 through capacity - 1. The following assignment results in an exception being thrown. The program usually terminates prematurely with a message like the one shown below.

```
x[8] = 4.5;  // This out-of-range index causes an exception
```

The program terminates prematurely (the output shows the index, which is 8 here).

```
java.lang.ArrayIndexOutOfBounds exception: 8
```

This might seem like a nuisance. However, without range checking, such out-of-range indexes could destroy the state of other objects in memory and cause difficult-to-detect bugs. More dramatically, your computer could "hang" or "crash." Even worse, with a workstation that runs all of the time, you could get an error that affects computer memory now, but won't crash the system until weeks later. However, in Java, you get the more acceptable occurrence of an `ArrayIndexOutOfBounds` exception while you are developing the code.

Self-Check

Use this initialization to answer the questions that follow:

```
int[] arrayOfInts = new int[100];
```

7-1 What type of element can be properly stored as elements in `arrayOfInts`?

7-2 How many integers may be properly stored as elements in `arrayOfInts`?

7-3 Which integer is used as the indearrayOfInts to access the first element in `arrayOfInts`?

7-4 Which integer is used as the indearrayOfInts to access the last element in `arrayOfInts`?

7-5 What is the value of `arrayOfInts[23]`?

7-6 Write code that stores 78 into the first element of `arrayOfInts`.

7-7 What would happen when this code executes? `ArrayOfInts[100] = 100;`

7.2 Array Processing with Determinate Loops

Programmers must frequently access consecutive array elements. For example, you might want to display all of the meaningful elements of an array containing test scores. The Java `for` loop provides a convenient way to do this.

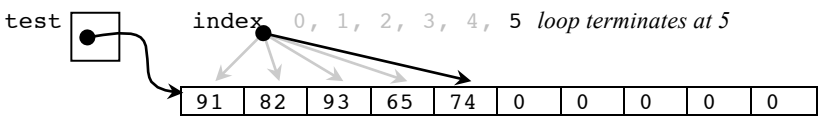
```
int[] test = new int[10];
test[0] = 91;
test[1] = 82;
test[2] = 93;
test[3] = 65;
test[4] = 74;

for (int index = 0; index < 5; index++) {
   System.out.println("test[" + index + "] == " + test[index]);
}
```

Output

```
test[0] == 91
test[1] == 82
test[2] == 93
test[3] == 65
test[4] == 74
```

Changing the `int` variable `index` from 0 through 4 provide accesses to all meaningful elements in the array referenced by `test`. This variable `index` acts both as the loop counter and as an array index inside the `for` loop (`test[index]`). With `index` serving both roles, the specific array

element accessed as `test[index]` depends on the value of `index`. For example, when `index` is `0`, `test[index]` references the first element in the array named `test`. When `index` is `4`, `test[index]` references the fifth element. Here is a more graphical view that shows the changing value of index.



Shortcut Array Initialization and the `length` Variable

Java also provides a quick and easy way to initialize arrays without using `new` or the capacity.

```
int[] test = { 91, 82, 93, 65, 74 };
```

The compiler sets the capacity of `test` to be the number of elements between `{` and `}`. The first value (91) is assigned to `test[0]`, the second value (82) to `test[1]`, and so on. Therefore, this shortcut array creation and assignment on one line are equivalent to these six lines of code for a completely filled array (no meaningless values).

```
int[] test = new int[5];
test[0] = 91;
test[1] = 82;
test[2] = 93;
test[3] = 65;
test[4] = 74;
```

This shortcut can be applied to all types.

```
double x[] = { 0.0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
char[] vowels = { 'a', 'e', 'i', 'o', 'u' };
String[] names = { "Tyler", "Angel", "Justice", "Reese" };
BankAccount[] accounts = { new BankAccount("Tyler", 100.00),
                           new BankAccount("Angel", 200.00),
                           new BankAccount("Justice", 300.00),
                           new BankAccount("Reese", 400.00)
                         };
```

The `length` variable stores the capacity of an array. It is often used to avoid out-of-range index exceptions. For example, the index range of the array `x` is `0` through `x.length - 1`. The capacity is referenced as the array name, a dot, and the variable named `length`. Do not use `()` after `length` as you would in a `String` message.

```
// Assert the capacities of the four arrays above
assertEquals(7, x.length);
assertEquals(5, vowels.length);
assertEquals(4, names.length);
assertEquals(4, accounts.length);
```

Argument/Parameter Associations

At some point, you will find it necessary to pass an array to another method. In this case, the parameter syntax requires `[]` and the correct type to mark the parameter can be matched to the array argument.

General Form: **Array parameters**

type **[ ]** *array-reference*

Example Array Parameters in method headings

```
public static void main(String[] args)
public double max(double[] x)
public boolean equal(double[] array1, double[] array2)
```

This allows array references to be passed into a method so that method has access to all elements in the array. For example, this method inspects the meaningful array elements (indexed from 0 through n - 1) to find the smallest value and return it.

```
public int min(int[] array, int n) {
  // Assume the first element is the smallest
  int smallest = array[0];
  // Inspect all other meaningful elements in array[1] through array[n-1]
  for (int index = 1; index < n; index++) {
    if (array[index] < smallest)
      smallest = array[index];
  }
  return smallest;
}
```

An array often stores fewer meaningful elements than its capacity. Therefore, the need arises to store the number of elements in the array that have been given meaningful values. In the previous code, `n` was used to limit the elements being referenced. Only the first five elements were considered to potentially be the smallest. Only the first five should have been considered. Without limiting the search to the meaningful elements (indexed as 0 through n - 1), would the smallest be 65 or would it be one of the 0s stored as one of the fifteen elements at the end that Java initialized to the default value of 0?

Consider the following test method that accidentally passes the array capacity as `test.length` (20) rather than the number of meaningful elements in the array (5).

```
@Test
public void testMin() {
  int[] test = new int[20];
  test[0] = 91;
  test[1] = 82;
  test[2] = 93;
  test[3] = 65;
  test[4] = 74;
  assertEquals(65, min(test, test.length));  // Should be 5
}
```

The assertion fails with this message:

```
java.lang.AssertionError: expected:<65> but was:<0>
```

If an array is "filled" with meaningful elements, the length variable can be used to process the array. However, since arrays often have a capacity greater than the number of meaningful elements, it may be better to use some separate integer variable with a name like `n` or `size`.

Messages to Individual Array Elements

The subscript notation must be used to send messages to individual elements. The array name must be accompanied by an index to specify the particular array element to which the message is sent.

***General Form:* Sending messages to individual array elements**

array-name **[***index***].***message-name***(***arguments)*

The *index* distinguishes the specific object the message is to be sent to. For example, the uppercase equivalent of `id[0]` (this element has the value `"Dylan"`) is returned with this expression:

```
names[0].toUpperCase(); // The first name in an array of Strings
```

The expression `names.toUpperCase()` is a syntax error because it attempts to find the uppercase version of the entire array, not one of its `String` elements. The `toUpperCase` method is not defined for standard Java array objects. On the other hand, `names[0]` does understand `toUpperCase` since `names[0]` is indeed a reference to a `String`. names is a reference to an array of Strings.

Now consider determining the total of all the balances in an array of `BankAccount` objects. The following test method first sets up a miniature database of four `BankAccount` objects. *Note:* A constructor call—with `new`—generates a reference to any type of object. Therefore this assignment

```
// A constructor first constructs an object, then returns its reference
account[0] = new BankAccount("Hall", 50.00);
```

first constructs a `BankAccount` object with the ID `"Hall"` and a balance of `50.0`. The reference to this object is stored in the first array element, `account[0]`.

```
@Test
public void testAssets() {
  BankAccount[] account = new BankAccount[100];
  account[0] = new BankAccount("Hall", 50.00);
  account[1] = new BankAccount("Small", 100.00);
  account[2] = new BankAccount("Ewall", 200.00);
  account[3] = new BankAccount("Westphall", 300.00);
  int n = 4;
  // Only the first n elements of account are meaningful, 96 are null
  double actual = assets(account, n);
  assertEquals(650.00, actual, 0.0001);
}
```

The actual return value from the assets method should be the sum of all account balances indexed from 0..n-1 inclusive, which is expected to be 650.0.

```
// Accumulate the balance of n BankAccount objects stored in account[]
public double assets(BankAccount[] account, int n) {
  double result = 0.0;
  for (int index = 0; index < n; index++) {
    result += account[index].getBalance();
  }
  return result;
}
```

Modifying Array Arguments

Consider the following method that adds the `incValue` to every array element. The test indicates that changes to the parameter `x` also modifies the argument `intArray`.

```
@Test
public void testIncrementBy() {
  int[] intArray = { 1, 5, 12 };
  increment(intArray, 6);
```

```
    assertEquals(7, intArray[0]);  // changing the elements of parameter x
    assertEquals(11, intArray[1]); // in increment is the same as changing
    assertEquals(18, intArray[2]); // intArray in this test method
  }

  public void increment(int[] x, int incValue) {
    for (int index = 0; index < x.length; index++)
      x[index] += incValue;
  }
```

To understand why this happens, consider the characteristics of reference variables.

A reference variable stores the location of an object, not the object itself. By analogy, a reference variable is like the address of a friend. It may be a description of where your friend is located, but it is not your actual friend. You may have the addresses of many friends, but these addresses are not your actual friends.

When the Java runtime system constructs an object with the `new` operator, memory for that object gets allocated somewhere in the computer's memory. The `new` operation then returns a reference to that newly constructed object. The reference value gets stored into the reference variable to the left of `=`. For example, the following construction stores the reference to a `BankAccount` object with `"Chris"` and `0.0` into the reference variable named `chris`.

```
BankAccount chris = new BankAccount("Chris", 0.00);
```

A programmer can now send messages to the object by way of the reference value stored in the reference variable named `chris`. The memory that holds the actual state of the object is stored elsewhere. Because you will use the reference variable name for the object, it is intuitive to think of `chris` as the object. However, `chris` is actually the reference to the object, which is located elsewhere.

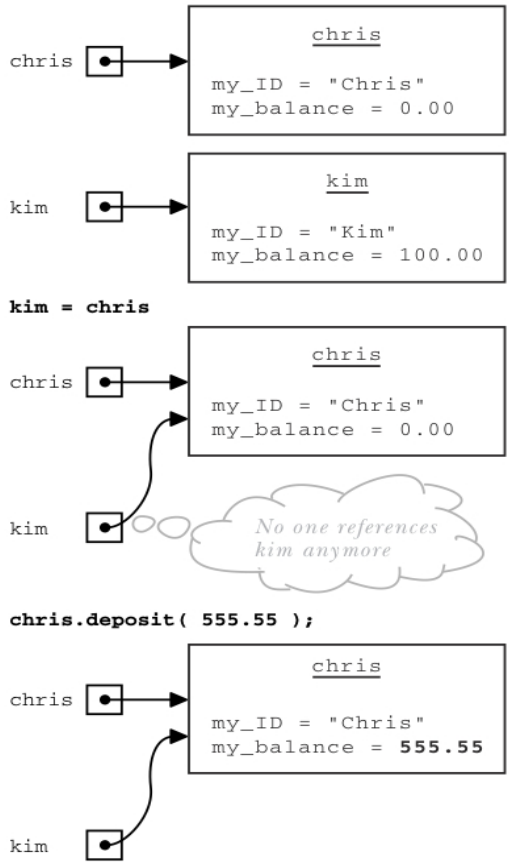
The following code mimics the same assignments that were made to the primitive variables above. The big difference is that the `deposit` message sent to `chris` actually modifies `kim`. This happens because both reference variables `chris` and `kim`—refer to the same object in memory after the assignment `kim = chris`. In fact, the object originally referred to by the reference variable named `kim` is lost forever. Once the memory used to store the state of an object no longer has any references, Java's garbage collector reclaims the memory so it can be reused later to store other new objects. This allows your computer to recycle memory that is no longer needed.

```
BankAccount chris = new BankAccount("Chris", 0.00);
BankAccount kim = new BankAccount("Kim", 100.00);
kim = chris;
// The values of the object were not assigned.
// Rather, the reference to chris was assigned to the reference variable kim.
// Now both reference variables refer to the same object.
System.out.println("Why does a change to 'chris' change 'kim'?");
chris.deposit(555.55);
System.out.println("Kim's balance was 0.00, now it is " + kim.getBalance());
```

Output

```
Why does a change to 'chris' change 'kim'?
Kim's balance was 0.00, now it is 555.55
```

Assignment statements copy the values to the right of `=` into the variable to the left of `=`. When the variables are primitive number types like `int` and `double`, the copied values are numbers. However, when the variables are references to objects, the copied values are the references to the objects in memory as illustrated in the following diagram.



After the assignment `kim = chris`, `kim` and `chris` both refer to the same object in memory. The state of the object is not assigned. Intead, the reference to the object is assigned. A message to either reference variable (`chris` or `kim`) accesses or modifies the same object, which now has the state of "Chris" and 555.55. An assignment of a reference value to another reference variable of the same type does not change the object itself. The state of an object can only be changed with messages designed to modify the state.

The big difference is that the `deposit` message to `chris` actually modified `kim`. This happens because both reference variables—`chris` and `kim`—refer to the same object in memory after the assignment `kim = chris`.

The same assignment rules apply when an argument is assigned to a parameter. In this method and test, chris and kim both refer to the same object.

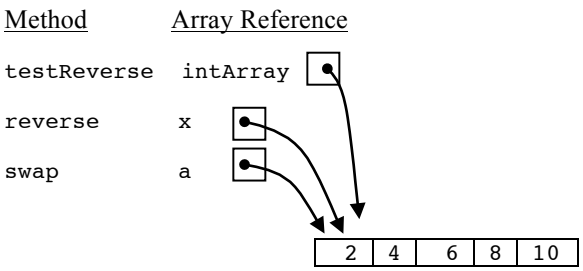
```
@Test
public void testAddToBalance() {
  BankAccount kim = new BankAccount("Chris", 0.00);
  assertEquals(0.0, kim.getBalance(), 0.0001);
  increment(kim);
  assertEquals(555.55, kim.getBalance(), 1e-14);
}

public void increment(BankAccount chris) {
  chris.deposit(555.55);
}
```

Java has one argument/parameter association. It is called pass by value. When an argument is assigned to a parameter, the argument's value is copied to the parameter. When the argument is a

primitive type such as `int` or `double`, the copied values are primitive numeric values or `char` values. No method can change the primitive arguments of another method. However, when an object reference is passed to a method, the value is a reference value. The argument is the location of the object in computer memory.

At that moment, the parameter is an alias (another name) for the argument. Two references to the same object exist. The parameter refers to the same object as the argument. This means that when a method modifies the parameter, the change occurs in the object referenced by the argument.

In this code that reverses the array elements, three reference variables reference the array of ints constructed in the test method.




```
@Test
public void testReverse() {
  int[] intArray = { 2, 4, 6, 8, 10 };
  reverse(intArray);
  assertEquals(10, intArray[0]); // was 2
  assertEquals(8, intArray[1]);  // was 4
  assertEquals(6, intArray[2]);  // was 6
  assertEquals(4, intArray[3]);  // was 8
  assertEquals(2, intArray[4]);  // was 10
}

// Reverse the array elements so x[0] gets exchanged with x[x.length-1],
// x[1] with x[x.length-2], x[2] with x[x.length-3], and so on.
public void reverse(int[] x) {
  int leftIndex = 0;
  int rightIndex = x.length - 1;
  while (leftIndex < rightIndex) {
    swap(x, leftIndex, rightIndex);
    leftIndex++;
    rightIndex--;
  }
}

// Exchange the two integers in the specified indexes
// inside the array referenced by a.
private void swap(int[] a, int leftIndex, int rightIndex) {
  int temp = a[leftIndex];      // Need to store a[leftIndex] before
  a[leftIndex] = a[rightIndex]; // a[leftIndex] gets erased in this assignment
  a[rightIndex] = temp;
}
```

Self-Check

7-8 Given the small change of < to <= in the for loop, describe what would happen when this method is called where the number of meaningful elements is n.

```
// Accumulate the balance of n BankAccount objects stored in account[]
public double assets(BankAccount[] account, int n) {
  double result = 0.0;
  for (int index = 0; index <= n; index++) {
    result += account[index].getBalance();
  }
  return result;
}
```

7-9 Write method `sameEnds` to return true if the integer in the first index equals the integer in the last index. This code must compile and the assertions must pass.

```
@Test public void testSameEnds() {
  int[] x1 = { 1, 2, 3, 4, 5 };
  int[] x2 = { 4, 3, 2, 1, 0, 1, 2, 4 };
  int[] x3 = { 5, 6 };
  int[] x4 = { 5, 5 };
  assertFalse(sameEnds(x1));
  assertTrue(sameEnds(x2));
  assertFalse(sameEnds(x3));
  assertTrue(sameEnds(x4));
}
```

7-10 Write method `swapEnds` that switches the end elements in an array of Strings. The following code must compile and the assertions must pass.

```
@Test public void testSwapEnds() {
  String[] strings = { "a", "b", "c", "x" };
  swapEnds(strings);
  assertEquals("x", strings[0]);
  assertEquals("b", strings[1]);
  assertEquals("c", strings[2]);
  assertEquals("a", strings[3]);
}

@Test public void testSwapEndsWhenLengthIsTwo() {
  String[] strings = { "a", "x" };
  swapEnds(strings);
  assertEquals("x", strings[0]);
  assertEquals("a", strings[1]);
}


@Test public void testSwapEndsWhenTooSmall() {
  String[] strings = { "a" };
  // There should be no exceptions thrown. Use guarded action.
  swapEnds(strings);
  assertEquals("a", strings[0]);
}
```

7-11 Write method for `accountsLargerThan` that takes an array of BankAccount references s and returns the number of accounts with a balance greater than the second argument of type double. The following test method must compile and the assertions must pass.

```
@Test
public void testAssets() {
  BankAccount[] account = new BankAccount[100];
  account[0] = new BankAccount("Hall", 50.00);
  account[1] = new BankAccount("Small", 100.00);
  account[2] = new BankAccount("Ewall", 200.00);
  account[3] = new BankAccount("Westphall", 300.00);
  int n = 4;

  int actual = studentsFun.accountsLargerThan(0.00, account, n);
  assertEquals(4, actual);
  actual = studentsFun.accountsLargerThan(50.00, account, n);
  assertEquals(3, actual);
  actual = studentsFun.accountsLargerThan(100.00, account, n);
  assertEquals(2, actual);
  actual = studentsFun.accountsLargerThan(200.00, account, n);
  assertEquals(1, actual);
  actual = studentsFun.accountsLargerThan(300.00, account, n);
  assertEquals(0, actual);
}
```

Answers to Self-Checks

7-1 `int` 7-2 100 7-3 0 7-4 99 7-5 0 7-6 `x[0] = 78;`

7-7 `ArrayIndexOutOfBounds` exception would terminate the program

7-8 There would be a `getBalance()` message sent to account[n+1] which is probably null. Program terminates

7-9
```
public boolean sameEnds(int[] array) {
  return array[0] == array[array.length-1];
}
```

7-10
```
private void swapEnds(String[] array) {
  if (array.length >= 2) {
    int rightIndex = array.length - 1;
    String temp = array[rightIndex];
    array[rightIndex] = array[0];
    array[0] = temp;
  }
}
```

7-11
```
public int accountsLargerThan(double amt, BankAccount[] account, int n) {
  int result = 0;
  for (int index = 0; index < n; index++) {
    if(account[index].getBalance() > amt)
      result ++;
  }
  return result;
}
```

Chapter 8

Search and Sort

Goals

This chapter begins by showing two algorithms used with arrays: selection sort and binary search. After studying this chapter, you will be able to

- understand how binary search finds elements more quickly than sequential search
- arrange array elements into ascending or descending order (sort them)
- Analyze the runtime of algorithms

8.1 Binary Search

The binary search algorithm accomplishes the same function as sequential search (see Chapter 8, "Arrays"). The binary search presented in this section finds things more quickly. One of the preconditions is that the collection must be sorted (a sorting algorithm is shown later).
The binary search algorithm works like this. If the array is sorted, half of the elements can be eliminated from the search each time a comparison is made. This is summarized in the following algorithm:

Algorithm: Binary Search, used with sorted arrays

```
while the element is not found and it still may be in the array {
    if the element in the middle of the array is the element being searched for
      store the reference and signal that the element was found so the loop can terminate
    else
      arrange it so that the correct half of the array is eliminated from further search
}
```

Each time the search element is not the element in the middle, the search can be narrowed. If the search item is less than the middle element, you search only the half that precedes the middle element. If the item being sought is greater than the middle element, search only the elements that are greater. The binary search effectively eliminates half of the array elements from the search. By contrast, the sequential search only eliminates one element from the search field with each comparison. Assuming that an array of strings is sorted in alphabetic order, sequentially searching for `"Ableson"` does not take long. `"Ableson"` is likely to be located near the front of the array elements. However, sequentially searching for `"Zevon"` takes much more time—especially if the array is very big (with millions of elements).

The sequential search algorithm used in the `indexOf` method of the previous chapter would have to compare all of the names beginning with A through Y before arriving at any names beginning with Z. Binary search gets to `"Zevon"` much more quickly. When an array is very large, binary search is much faster than sequential search. The binary search algorithm has the following preconditions:

1. The array must be sorted (in ascending order, for now).
2. The indexes that reference the first and last elements must represent the entire range of meaningful elements.

The index of the element in the middle is computed as the average of the first and last indexes. These three indexes—named `first`, `mid`, and `last`—are shown below the array to be searched.

```
int n = 7;
String[] name = new String[n];
name[0] = "ABE";
name[1] = "CLAY";
name[2] = "KIM";
name[3] = "LAU";
name[4] = "LISA";
name[5] = "PELE";
name[6] = "ROY";
// Binary search needs several assignments to get things going
int first = 0;
int last = n - 1;
int mid = (first + last) / 2;
String searchString = "LISA";
// -1 will mean that the element has not yet been found
int indexInArray = -1;
```

Here is a more refined algorithm that will search as long as there are more elements to look at and the element has not yet been found.

Algorithm: Binary Search *(more refined, while still assuming that the items have been sorted)*

while indexInArray is -1 and there are more array elements to look through {
  if searchString is equal to name[mid] then
    let indexInArray = mid *// This indicates that the array element equaled searchString*
  else if searchString alphabetically precedes name[mid]
    eliminate mid . . . last elements from the search
  else
    eliminate first . . . mid elements from the search
  mid = (first + last) / 2; *// Compute a new mid for the next loop iteration (if there is one)*
}
// At this point, indexInArray is either -1, indicating that searchString was not found,
// or in the range of 0 through n - 1, indicating that searchString was found.

As the search begins, one of three things can happen (the code is searching for a `String` that equals `searchString`):

1. The element in the middle of the array equals `searchString`. The search is complete. Store `mid` into `indexInArray` to indicate where the `String` was found.
2. `searchString` is less than (alphabetically precedes) the middle element. The second half of the array can be eliminated from the search field (`last = mid - 1`).
3. `searchString` is greater than (alphabetically follows) the middle element. The first half of the array can be eliminated from the search field (`first = mid + 1`).

In the following code, if the `String` being searched for is not found, `indexInArray` remains –1. As soon as an array element is found to equal `searchString`, the loop terminates. The

second part of the loop test stops the loop when there are no more elements to look at, when `first` becomes greater than `last`, or when the entire array has been examined.

```
    // Binary search if searchString
    // is not found and there are more elements to compare.
    while (indexInArray == -1 && (first <= last)) {
      // Check the three possibilities
      if (searchString.equals(name[mid]))
        indexInArray = mid; // 1. searchString is found
      else if (searchString.compareTo(name[mid]) < 0)
        last = mid - 1; // 2. searchString may be in first half
      else
        first = mid + 1; // 3. searchString may be in second half

      // Compute a new array index in the middle of the search area
      mid = (first + last) / 2;
    } // End while

  // indexInArray now either is -1 to indicate the String is not in the array
  // or when indexInArray >= 0 it is the index of the first equal string found.
```

At the beginning of the first loop iteration, the variables `first`, `mid`, and `last` are set as shown below. Notice that the array is in ascending order (binary search won't work otherwise).

Array and binary search indexes before comparing `searchString` (`"LISA"`) to `name[mid]` (`"LAU"`):

```
name[0]   "ABE"  ⇐ first == 0
name[1]   "CLAY"
name[2]   "KIM"
name[3]   "LAU"  ⇐ mid == 3
name[4]   "LISA"
name[5]   "PELE"
name[6]   "ROY"  ⇐ last == 6
```

After comparing `searchString` to `name[mid]`, `first` is increased from `0` to `mid + 1`, or `4`; `last` remains `6`; and a new `mid` is computed as (4 + 6) / 2 = `5`.

~~`name[0]  "ABE"`~~
~~`name[1]  "CLAY"`~~
~~`name[2]  "KIM"`~~
~~`name[3]  "LAU"`~~

Because "LISA" is greater than name[mid], the objects name[0] through name[3] no longer need to be searched through and can be eliminated from subsequent searches. That leaves only three possibilities.

```
name[4]   "LISA" ⇐   first == 4
name[5]   "PELE" ⇐     mid == 5
name[6]   "ROY"  ⇐    last == 6
```

With `mid == 5`, `"LISA".compareTo("PELE") < 0` is true. So `last` is decreased (5 - 1 = 4), `first` remains `4`, and a new `mid` is computed as `mid = (4 + 4) / 2 = 4`.

~~`name[0]  "ABE"`~~
~~`name[1]  "CLAY"`~~
~~`name[2]  "KIM"`~~
~~`name[3]  "LAU"`~~

```
name[4]   "LISA" ⇐ mid == 4   ⇐ first == 4   ⇐ last == 4
```

~~`name[5]  "PELE"`~~
~~`name[6]  "ROY"`~~

Because "LISA" is less than name[mid], eliminate name[6].

Now name[mid] does equal searchString ("LISA".equals("LISA")), so `indexInArray = mid`. The loop terminates because `indexInArray` is no longer -1. The following code after the loop and the output confirm that `"LISA"` was found in the array.

```
if (indexInArray == -1)
  System.out.println(searchString + " not found");
else
  System.out.println(searchString + " found at index " + indexInArray);
```

Output

```
LISA found at index 4
```

Terminating when searchName Is Not Found

Now consider the possibility that the data being searched for is not in the array; if `searchString` is `"DEVON"`, for example.

```
// Get the index of DEVON if found in the array
String searchName = "DEVON";
```

This time the values of `first`, `mid`, and `last` progress as follows:

	first	***mid***	***last***	***Comment***
#1	0	3	6	Compare "DEVON" to "LAU"
#2	0	1	2	Compare "DEVON" to "CLAY"
#3	2	2	2	Compare "DEVON" to "KIM"
#4	2	2	1	`first <= last` is false—the loop terminates

When the `searchString` (`"DEVON"`) is not in the array, `last` becomes less than `first` (`first > last`). The two indexes have crossed each other. Here is another trace of binary search when the searched for element is not in the array.

```
                      #1          #2           #3                    #4
name[0]   "ABE"     ⇐ first     ⇐ first
name[1]   "CLAY"                ⇐ mid                              last
name[2]   "KIM"                 ⇐ last       ⇐ first, mid, last    first
name[3]   "LAU"     ⇐ mid
name[4]   "LISA"
name[5]   "PELE"
name[6]   "ROY"     ⇐ last
```

After `searchString` ("DEVON") is compared to `name[2]` (`"KIM"`), no further comparisons are necessary. Since `DEVON` is less than `KIM`, `last` becomes `mid` - 1, or 1. The new `mid` is computed to be 2, but it is never used as an index. This time, the second part of the loop test terminates the loop.

```
while(indexInArray == -1 && (first <= last))
```

Since `first` is no longer less than or equal to `last`, `searchString` cannot be in the array. The `indexInArray` remains `-1` to indicate that the element was not found.

Comparing Running Times

The binary search algorithm can be more efficient than the sequential search algorithm. Whereas sequential search only eliminates one element from the search per comparison, binary search eliminates half of the elements for each comparison. For example, when the number of elements

(n) == 1,024, a binary search eliminates 512 elements from further search in the first comparison, 256 during the second comparison, then 128, 64, 32, 16, 4, 2, and 1.

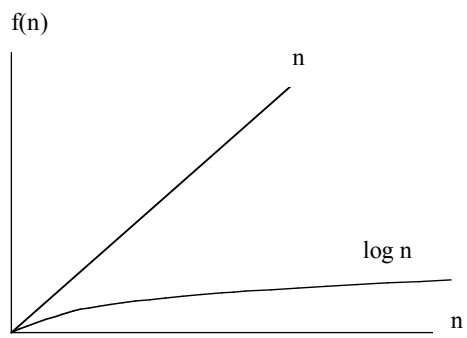
When n is small, the binary search is not much faster than sequential search. However, when n gets large, the difference in the time required to search for something can make the difference between selling the software and having it flop. Consider how many comparisons are necessary when n grows by powers of two. Each doubling of n would require potentially twice as many loop iterations for sequential search. However, the same doubling of n would require potentially only one more comparison for binary search.

The Maximum Number of Comparisons during Two Different Search Algorithms

Power of 2	n	Sequential Search	Binary Search
2^2	4	4	2
2^4	16	16	4
2^8	256	256	8
2^{12}	4,096	4,096	12
2^{24}	16,777,216	16,777,216	24

As n gets very large, sequential search has to do a lot more work. The numbers above represent the maximum number of iterations to find an element or to realize it is not there. The difference between 24 comparisons and almost 17 million comparisons is quite dramatic even on a fast computer.

In general, as the number of elements to search (n) doubles, binary search requires only one iteration to eliminate half of the elements from the search. The growth of this function is said to be logarithmic. The following graph illustrates the difference between linear search and binary search as the size of the array grows.



Self-Check

8-1 Give at least one precondition for a successful binary search.

8-2 What is the maximum number of comparisons (approximately) performed on a list of 1,024 elements during a binary search? (Hint: After one comparison, only 512 array elements need be searched; after two searches, only 256 elements need be searched, and so on.)

8-3 During a binary search, what condition signals that the search element does not exist in an array?

8-4 What changes would be made to the binary search when the elements are sorted in descending order?

8.2 One Sorting Algorithm

The elements of a collection are often arranged into either ascending or descending order through a process known as **sorting**. To sort an array, the elements must be compared. For `int` and `double`, < or > suffices. For `String`, `Integer`, and `BankAccount` objects, the `compareTo` method is used.

There are many sorting algorithms. Even though others are more efficient (run faster), the relatively simple selection sort is presented here. The goal here is to arrange an array of integers into ascending order, the natural ordering of integers.

Object Name	Unsorted Array	Sorted Array
`data[0]`	`76.0`	`62.0`
`data[1]`	`91.0`	`76.0`
`data[2]`	`100.0`	`89.0`
`data[3]`	`62.0`	`91.0`
`data[4]`	`89.0`	`100.0`

With the selection sort algorithm, the largest integer must end up in `data[n - 1]` (where n is the number of meaningful array elements). The smallest number should end up in `data[0]`. In general, an array `x` of size `n` is sorted in ascending order if `x[j] <= x[j + 1]` for `j = 0` to `n-2`.

The selection sort begins by locating the smallest element in the array by searching from the first element (`data[0]`) through the last (`data[4]`). The smallest element, `data[2]` in this array, is then swapped with the top element, `data[0]`. Once this is done, the array is sorted at least through the first element.

top == 0	Before	After	Sorted
`data[0]`	***76.0***	***62.0***	⇐
`data[1]`	`91.0`	`91.0`	
`data[2]`	`100.0`	`100.0`	
`data[3]`	***62.0***	***76.0***	
`data[4]`	`89.0`	`89.0`	

Placing the Largest Value in the "Top" Position (index 0)

The task of finding the smallest element is accomplished by examining all array elements and keeping track of the index with the smallest integer. After this, the smallest array element is swapped with `data[0]`. Here is an algorithm that accomplishes these two tasks:

Algorithm: Finding the smallest in the array and switching it with the topmost element

(a) top = 0

// At first, assume that the first element is the smallest

(b) indexOfSmallest = top

// Check the rest of the array (data[top + 1] through data[n - 1])

(c) for index ranging from top + 1 through n - 1

(c1) if data[index] < data[indexOfSmallest]

indexOfSmallest = index

// Place the smallest element into the first position and place the first array

// element into the location where the smallest array element was located.

(d) swap data[indexOfSmallest] with data[top]

The following algorithm walkthrough shows how the array is sorted through the first element. The smallest integer in the array will be stored at the "top" of the array—`data[0]`. Notice that `indexOfSmallest` changes only when an array element is found to be less than the one stored in `data[indexOfSmallest]`. This happens the first and third times step c1 executes.

Step	`top`	`indexOf Smallest`	`index`	`[0]`	`[1]`	`[2]`	`[3]`	`[4]`	`n`
	?	?	?	76.0	91.0	100.0	62.0	89.0	5
(a)	0	"	"	"	"	"	"	"	"
(b)	"	0	"	"	"	"	"	"	"
(c)	"	"	1	"	"	"	"	"	"
(c1)	"	***1***	"	"	"	"	"	"	"
(c)	"	"	2	"	"	"	"	"	"
(c1)	"	"	"	"	"	"	"	"	"
(c)	"	"	3	"	"	"	"	"	"
(c1)	"	***2***	"	"	"	"	"	"	"
(c)	"	"	4	"	"	"	"	"	"
(c1)	"	"	"	"	"	"	"	"	"
(c)	"	"	5	"	"	"	"	"	"
(d)	"	"	"	***62.0***	"	"	***76.0***	"	"

This algorithm walkthrough shows `indexOfSmallest` changing twice to represent the index of the smallest integer in the array. After traversing the entire array, the smallest element is swapped with the top array element. Specifically, the preceding algorithm swaps the values of the first and fourth array elements, so `62.0` is stored in `data[0]` and `76.0` is stored in `data[3]`. The array is now sorted through the first element!

The same algorithm can be used to place the second smallest element into `data[1]`. The second traversal must begin at the new "top" of the array—index 1 rather than 0. This is accomplished by incrementing `top` from 0 to 1. Now a second traversal of the array begins at the second element rather than the first. The smallest element in the unsorted portion of the array is swapped with the second element. A second traversal of the array ensures that the first two elements are in order. In this example array, `data[3]` is swapped with `data[1]` and the array is sorted through the first two elements.

top == 1	Before	After	Sorted
`data[0]`	`62.0`	`62.0`	⇐
`data[1]`	***`91.0`***	***`76.0`***	⇐
`data[2]`	`100.0`	`100.0`	
`data[3]`	***`76.0`***	***`91.0`***	
`data[4]`	`89.0`	`89.0`	

This process repeats a total of n - 1 times.

top == 2	Before	After	Sorted
`data[0]`	`62.0`	`62.0`	⇐
`data[1]`	`76.0`	`76.0`	⇐
`data[2]`	***`100.0`***	***`89.0`***	⇐
`data[3]`	`91.0`	`91.0`	
`data[4]`	***`89.0`***	***`100.0`***	

An element may even be swapped with itself.

top == 3	Before	After	Sorted
data[0]	62.0	62.0	⇐
data[1]	76.0	76.0	⇐
data[2]	89.0	89.0	⇐
data[3]	***91.0*** ⟵	⟶ ***91.0***	⇐
data[4]	100.0	100.0	

When `top` goes to `data[4]`, the outer loop stops. The last element need not compared to anything. It is unnecessary to find the smallest element in an array of size 1. This element in `data[n - 1]` must be the largest (or equal to the largest), since all of the elements preceding the last element are already sorted in ascending order.

top == 3 and 4	Before	After	Sorted
data[0]	62.0	62.0	⇐
data[1]	76.0	***76.0***	⇐
data[2]	89.0	89.0	⇐
data[3]	91.0	91.0	⇐
data[4]	100.0	100.0	⇐

Therefore, the outer loop changes the index `top` from 0 through n - 2. The loop to find the smallest index in a portion of the array is nested inside a loop that changes `top` from 0 through n - 2 inclusive.

Algorithm: Selection Sort

```
for top ranging from 0 through n - 2  {
   indexOfSmallest = top
   for index ranging from top + 1 through n - 1    {
       if data[indexOfSmallest] < data[index] then
         indexOfSmallest = index
   }
   swap  data[indexOfSmallest] with data[top]
}
```

Here is the Java code that uses selection sort to sort the array of numbers shown. The array is printed before and after the numbers are sorted into ascending order.

```
double[] data = { 76.0, 91.0, 100.0, 62.0, 89.0 };
int n = data.length;

System.out.print("Before sorting: ");
for(int j = 0; j < data.length; j++)
  System.out.print(data[j] + " ");
System.out.println();

int indexOfSmallest = 0;

for(int top = 0; top < n - 1; top++) {
  // First assume that the smallest is the first element in the subarray
  indexOfSmallest = top;
```

```
      // Then compare all of the other elements, looking for the smallest
      for(int index = top + 1; index < data.length; index++)
      { // Compare elements in the subarray
        if(data[index] < data[indexOfSmallest])
          indexOfSmallest = index;
      }

      // Then make sure the smallest from data[top] through data.size
      // is in data[top]. This message swaps two array elements.
      double temp = data[top]; // Hold on to this value temporarily
      data[top] = data[indexOfSmallest];
      data[indexOfSmallest] = temp;
    }
    System.out.print(" After sorting: ");
    for (int j = 0; j < data.length; j++)
      System.out.print(data[j] + " ");
    System.out.println();
```

Output

```
Before sorting: 76.0 91.0 100.0 62.0 89.0
 After sorting: 62.0 76.0 89.0 91.0 100.0
```

Sorting an array usually involves elements that are more complex. The sorting code is most often located in a method. This more typical context for sorting will be presented later.

This selection sort code arranged the array into ascending numeric order. Most sort routines arrange the elements from smallest to largest. However, with just a few simple changes, any primitive type of data (such as `int`, `char`, and `double`) may be arranged into descending order using the > operator.

```
if(data[index] < data[indexOfSmallest])
   indexOfSmallest = index;
```

becomes

```
if(data[index] > data[indexOfLargest])
  indexOfLargest = index;
```

Only primitive types can be sorted with the relational operators < and >. Arrays of other objects, `String` and `BankAccount` for example, have a `compareTo` method to check the relationship of one object to another.

Self-Check

8-5 Alphabetizing an array of strings requires a sort in which order, ascending or descending?

8-6 If the smallest element in an array already exists as `first`, what happens when the swap function is called for the first time (when `top` = 0)?

8-7 Write code that searches for and stores the largest element of array `x` into `largest`. Assume that all elements from `x[0]` through `x[n - 1]` have been given meaningful values.

Answers to Self-Check Questions

8-1 The array is sorted.

8-2 1,024; 512; 256; 128; 64; 32; 16; 8; 4; 2; 1 == 11

8-3 When `first` becomes greater than `last`.

8-4 Change the comparison from less than to greater than.

```
if(searchString.compareTo(str[mid]) > 0)
  last = mid - 1;
else
   first= mid + 1;  // ...
```

8-5 Ascending

8-6 The first element is swapped with itself.

8-7
```
int largest = x[0];
for(int j = 0; j < n; j++) {
  if(x[j] > largest)
     largest = x[j];
}
```

Chapter 9

Classes with Instance Variables

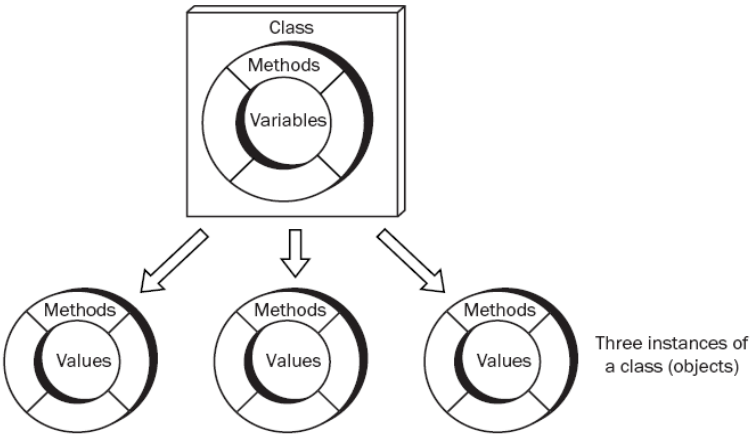
Goals

- Implement Java Classes as a set of methods and variables
- Experience designing and testing a class that is part of a large system

9.1 Constructing Objects from Classes

Object-oriented programs use objects constructed from many different classes. They may be established Java classes that are part of the download, classes bought from other software developers, classes downloaded for free, or classes designed by programmers to fulfill the needs of a particular application. A class provides a blueprint for constructing objects, and defines the messages that will be available to instances of each class. The class also defines the values that are encapsulated in every object as the object's state.

One class constructing three objects, each with its own set of values (state)



Every Java class has methods that represent the messages each object will understand. Each object of the class has its own set of instance variables to store the values contained in each object. The collection of instance variables is also known as the state of the object.

Methods and Data Together

All classes have these things in common:

- private instance variables that store the state of the objects
- constructors that initialize the state
- methods to modify the state of objects
- methods to provide access to the current state of objects

Java classes begin with `public class` followed by the class name. The instance variables and methods follow within a set of matching curly braces. The methods and state should have some sort of meaningful connection.

***Simplified General Form:* A Java class**

```
public class class-name {

  // Instance variables (every instance of this class will get its own)
  private variable declaration;
  private variable initialization;

  // Constructor(s) (methods with the same name as the class and no return type)
  public class-name(parameters) {
    // ...
  }

  // Any number of methods
  public return-type method-name-1(parameters) {
    // ...
  }
}
```

Here is a simplified version of the `BankAccount` class. The two instance variables `ID` and `balance` are available to all methods of the class.

```
// This class models a minimal bank account.
public class BankAccount {

  // Instance variables--every BankAccount object will have its own values.
  private String ID;
  private double balance;

  // Initialize instance variables during construction.
  public BankAccount(String initialID, double initialBalance) {  ❶
    ID = initialID;
    balance = initialBalance;
  }

  public void deposit(double depositAmount) {                    ❷
     balance = balance + depositAmount;
  }

  public void withdraw(double withdrawalAmount) {                ❸
     balance = balance - withdrawalAmount;
  }

  public String getID() {                                        ❹
    return ID;
  }
  public double getBalance() {                                   ❺
    return balance;
  }
}
```

With the class stored in a file, it can be used as a blueprint to construct many objects. Each object will have its own `ID` and balance. Each object will understand the withdraw, deposit, getID, and getBalance methods. In the following program, the numbers (❶ for example) indicate which method will execute when the message is sent.

For example, ❷ represents transfer of control from the main method to the deposit method in the BankAccount class.

```
@Test
public void testToDemonstrateControlFlow() {
  BankAccount acctOne = new BankAccount("01543C", 100.00);      ❶
  acctOne.deposit(50.0);                                        ❷
  acctOne.withdraw(25.0);                                       ❸
  assertEquals("01543C", acctOne.getID());                      ❹
  assertEquals(125.0, acctOne.getBalance(), 1e-14);             ❺
}
```

Instance Variables

In this first example of a type implemented as a Java class, each `BankAccount` object stores data to represent a simple account at a bank. Each `BankAccount` object stores some unique identification ID and an account balance. `BankAccount` methods include making deposits, making withdrawals, and accessing the ID and the current balance.

The private instance variables represent the state. `BankAccount` has two private instance variables: `ID` (a `String`) and `balance` (a `double`). Every `BankAccount` object remembers its own ID and its own current balance.

Notice that the instance variables are not declared within a method. They are declared within the set of curly braces that bounds the class. This means that the instance variables will be accessible throughout the class, and every method will have access to them.

If you look at the `BankAccount` class again, you will notice that every method references at least one of the instance variables. Also, each instance variable is accessed by at least two methods (both the constructor BankAccount and getID need ID).

Because the instance variables are declared `private`, programs using instances of the class cannot access the instance variables directly. This is good. The class safely encapsulated the state, which was initialized by the constructor (described below). The only way to then change or access the state of an object is through public methods.

Constructors

The `BankAccount` class shows that all `BankAccount` method headings are public. They also have return types (including **`void`** to mean return nothing). Some have parameters. However, do you notice something different about the method named `BankAccount`?

The `BankAccount` method has no return type. It also has the same name as the class! This special method is known as a **constructor,** because it is the method called when objects are constructed. When a constructor is called, memory is allocated for the object. Then, the instance variables are initialized, often with the arguments to the constructor. Here are some object constructions that result in executing the class's constructor while passing values:

```
new String("An initial part of this object's state");
new BankAccount("Charlie", 10.00);
```

Constructor parameters often initialize the private instance variables. The constructor returns a reference to the new object. This reference value can then be assigned to an object reference of the same type. That is why you often see the class name on both sides of the assignment operator =. For example, the following code constructs a `BankAccount` object with an initial ID of `"Phoenix"` and an initial balance of `507.34`. After the constructor has been called, the reference to this new `BankAccount` object is assigned to the reference variable named one.

```
BankAccount one = new BankAccount("Phoenix", 507.34);
```

The following code implements `BankAccount`'s two-parameter constructor:

```
// This constructor initializes the values of the instance variables
// using the arguments use when objects are constructed.
public BankAccount(String accountID, double initialBalance) {
   ID = accountID;
   balance = initialBalance;
}
```

This method executes whenever a `BankAccount` gets constructed with two arguments (a `String` followed by a double). For example, in the following code, the ID `"Jessie"` is passed to the parameter `ID`, which in turn is assigned to the private instance variable `ID`. The starting balance of `500.00` is also passed to the parameter named `initialBalance`, which in turn is assigned to the private instance variable `balance`.

```
BankAccount anAccount = new BankAccount("Jessie", 500.00);
```

Some methods provide access to private instance variables. They are sometimes called “getters”, because the method "gets" the value of an instance variable (and they usually begin with `get`). These methods often simply return the value of an instance variable with the `return` statement. Getter methods are necessary because the instance variables are not directly accessible when they are declared `private`.

```
public String getID() {
  return ID;
}

public double getBalance() {
  return balance;
}
```

To get the ID and balance, send the object separate getID and getBalance messages.

```
@Test
public void showMessagesWayAhead() {
  BankAccount anAccount = new BankAccount("Jessie", 500.00);
  assertEquals("Jessie", anAccount.getID());
  assertEquals(500.00, anAccount.getBalance(), 1e-14);
}
```

The state of an object can change. Some methods are designed to modify the values of the instance variables. Both deposit and withdraw change the state.

```
public void deposit(double depositAmount) {
  balance = balance + depositAmount;
}

public void withdraw(double withdrawalAmount) {
 balance = balance - withdrawalAmount;
}
```

These two simple test methods assert the changing state of an object.

```
@Test
public void testDepositWithPositiveAmount() {
  BankAccount anAccount = new BankAccount("Jessie", 500.00);
  anAccount.deposit(123.45);
  assertEquals(623.45, anAccount.getBalance(), 1e-14);
}

@Test
public void testWithdrawWithPositiveAmount() {
  BankAccount anAccount = new BankAccount("Jessie", 500.00);
  anAccount.withdraw(123.45);
  assertEquals(376.55, anAccount.getBalance(), 1e-14);
}
```

Self-Check

Use the following `SampleClass` to answer the Self-Check question that follows.

```
// A class that has no meaning other than to show the syntax of a class.
public class SampleClass {

  // Instance variables
  private int first;
  private int second;

  public SampleClass(int initalFirst, int initialSecond) {
    first = initalFirst;
    second = initialSecond;
  }

  public int getFirst() {
    return first;
  }

  public int getSecond() {
    return second;
  }

  public void change(int amount) {
    first = first + amount;
    second = second - amount;
  }
} // End SampleClass
```

9-1 Fill in the blanks that would make the assertions pass.

```
// A unit test to test class SampleClass
import static org.junit.Assert.*;
import org.junit.Test;

public class SampleClassTest {

  @Test
  public void testGetters() {
    SampleClass sc1 = new SampleClass(1, 4);
    SampleClass sc2 = new SampleClass(3, 5);
    assertEquals(_____, sc1.getFirst());
    assertEquals(_____, sc1.getSecond());
    assertEquals(_____, sc2.getFirst());
    assertEquals(_____, sc2.getSecond());
  }

  @Test
  public void testChange() {
    SampleClass sc1 = new SampleClass(1, 4);
    SampleClass sc2 = new SampleClass(3, 5);
    sc1.change(7);
    sc2.change(-3);
    assertEquals(_____, sc1.getFirst());
    assertEquals(_____, sc1.getSecond());
    assertEquals(_____, sc2.getFirst());
    assertEquals(_____,sc2.getSecond());
  }
}
```

Use this Java class to answer the questions that follow.

```java
// A class to model a simple library book.
public class LibraryBook {

  // Instance variables
  private String author;
  private String title;
  private String borrower;

  // Construct a LibraryBook object and initialize instance variables
  public LibraryBook(String initTitle, String initAuthor) {
    title = initTitle;
    author = initAuthor;
    borrower = null; // When borrower == null, no one has the book
  }

  // Return the author.
  public String getAuthor() {
    return author;
  }

  // Return the borrower's name if the book has been checked out or null if not
  public String getBorrower() {
    return borrower;
  }

  // Records the borrower's name
  public void borrowBook(String borrowersName) {
    borrower = borrowersName;
  }

  // The book becomes available. When null, no one is borrowing it.
  public void returnBook() {
    borrower = null;
  }
}
```

9-2 What is the name of the type above?

9-3 What is the name of the constructor?

9-4 Except for the constructor, name all of the methods.

9-5 `getBorrower` returns a reference to what type?

9-6 `borrowBook` returns a reference to what type?

9-7 What type argument must be part of all `borrowBook` messages?

9-8 How many arguments are required to construct one `LibraryBook` object?

9-9 Write the code to construct one `LibraryBook` object using your favorite book and author.

9-10 Send the message that borrows your favorite book. Use your own name as the borrower.

9-11 Write the message that reveals the name of the person who borrowed your favorite book (or `null` if no one has borrowed it).

9-12 Which of the following two assertions will pass, a, b, or both?

```java
@Test
public void testGetters() {
  LibraryBook book1 = new LibraryBook("C++", "Michael Berman");
  assertEquals(null, book1.getBorrower()); // a.
  book1.borrowBook("Sam Mac");
  assertEquals("Sam Mac", book1.getBorrower()); // b.
}
```

9-13 Write method `getTitle` that returns the title of any `LibraryBook` object.

9-14 Fill in the blanks so the assertions pass.

```
@Test
public void testGetters() {
  LibraryBook book1 = new LibraryBook("C++", "Michael Berman");
  assertEquals([              ], book1.getAuthor());
  assertEquals([              ], book1.getBorrower());
}
```

9-15 Write method `isAvailable` as if it were inside the `LibraryBook` class to return `false` if a `LibraryBook` is not borrowed or true if the borrower is `null`. Use `==` to compare `null` to an object reference.

9-16 Fill in the blanks in this test method to verify `getTitle` works so all assertions pass.

```
@Test
public void isAvailable() {
   LibraryBook book1 = new LibraryBook("C++", "Berman");
   LibraryBook book2 = new LibraryBook("C#", "Stepp");
   assert[       ](book1.isAvailable());
   assert[       ](book2.isAvailable());

   book1.borrowBook("Sam ");
   book2.borrowBook("Li");
   assert[       ](book1.isAvailable());
   assert[       ](book2.isAvailable());
}
```

Overriding toString

Each class should have its own `toString` method so the state of the object can be visually inspected. Java is designed such that all classes extend a class named `Object` (or each class extends a class that extends the `Object` class). This means all Java classes inherit the eleven methods of `Object`, one of which is `toString`. Doing nothing to a new class allows `toString` messages to invoke the `toString` method of class `Object`. The return string is the name of the class followed by `@` followed by a code written in hexadecimal (base 16 where 10 is A and 15 is F).

```
LibraryBook book1 = new LibraryBook("C++", "Berman");
System.out.println(book1.toString());
```

Output

```
LibraryBook@e4457d
```

To get a more meaningful `toString` that shows the current state of any object, you can override the `toString` method of class `Object` with the same method signature.

```
public String toString() {
  return title + ", borrower: " + borrower;
}
```

With the `toString` method of `Object` overridden to reflect the new type, the output better represents the state of the object.

```
@Test
public void testToString() {
  LibraryBook book1 = new LibraryBook("C++", "Michael A. Berman");
  LibraryBook book2 = new LibraryBook("Java", "Rick Mercer");
  book2.borrowBook("Sam Mac");
  assertEquals("C++, borrower: null", book1.toString());
  assertEquals("Java, borrower: Sam Mac", book2.toString());
}
```

Self Check

9-17 Add a `toString` method for the `BankAccount` class to show the ID followed by a blank space and the current balance. You will need the instance variables in `BankAccount`.

```
public class BankAccount {
  private String ID;
  private double balance;

  public BankAccount(String initID, double initBalance) {
    ID = initID;
    balance = initBalance;
  }
  // Add toString as if it were here
}
```

Naming Conventions

A method that modifies the state of an object is typically given a name that indicates its behavior. This is easily accomplished if the designer of the class provides a descriptive name for the method. The method name should describe—as best as possible—what the method actually does. It should also help to distinguish modifying methods from accessing methods. Use verbs to name modifying methods: `withdraw`, `deposit`, `borrowBook`, and `returnBook`, for example. Give accessing methods names to indicate that the messages will return some useful information about the objects: `getBorrower` and `getBalance`, for example. Above all, always use intention-revealing identifiers to accurately describe what the method does. For example, don't use `foo` as the name of a method that withdraws money.

`public` or **`private`**?

One of the considerations in the design of any class is declaring methods and instance variables with the most appropriate access mode, either `public` or `private`. Whereas programs outside the class can access the public methods of a class, the **`private`** instance variables are only known in the class methods. For example, the `BankAccount` instance variable named `balance` is known only to the methods of the class. On the other hand, any method declared `public` is known wherever the object was declared.

Access Mode	Where the Identifier Can Be Accessed (where the identifier is visible)
`public`	In all parts of the class and anywhere an instance of the class is declared
`private`	Only in the same class

Although instance variables representing state could be declared as `public`, it is highly recommended that all instance variables be declared as `private`. There are several reasons for this. The consistency helps simplify some design decisions. More importantly, when instance variables are made `private`, the state can be modified only through a method. This prevents other code from indiscriminately changing the state of objects. For example, it is impossible to accidentally make a credit to `acctOne` like this:

```
BankAccount acctOne = new BankAccount("Mine", 100.00);
// A compiletime error occurs: attempting to modify private data
acctOne.balance = acctOne.balance + 100000.00;  // <- ERROR
```

or a debit like this:

```
// A compiletime error occurs at this attempt to modify private data
acctOne.balance = acctOne.balance - 100.00;  // <- ERROR
```

This represents a widely held principle of software development—data should be hidden. Making instance variables `private` is one characteristic of a well-designed class.

Answers to Self-Check

9-1
```
SampleClass sc2 = new SampleClass(3, 5);
assertEquals(__1__, sc1.getFirst());
assertEquals(__4__, sc1.getSecond());
assertEquals(__3__, sc2.getFirst());
assertEquals(__5__, sc2.getSecond());
sc2.change(-3);
assertEquals(__8__, sc1.getFirst());
assertEquals(__-3__, sc1.getSecond());
assertEquals(__0__, sc2.getFirst());
assertEquals(__8__, sc2.getSecond());
```

9-2 type: `LibraryBook`

9-3 constructor: `LibraryBook`

9-4 `LibraryBook` (constructor) `getAuthor getBorrower borrowBook returnBook`

9-5 `String`

9-6 nothing, it is a void return type.

9-7 `String`

9-8 two (both `String`)

9-9 `LibraryBook aBook = new LibraryBook("Computing Fundamentals", "Rick Mercer");`

9-10 `aBook.borrowBook("Kim");`

9-11 `aBook.getBorrower();`

9-12 both a and b pass

9-13
```
public String getTitle() {
  return title;
}
```

9-14
```
@Test
public void testGetters() {
  LibraryBook book1 = new LibraryBook("C++", "Michael Berman");
  assertEquals("Michael Berman", book1.getAuthor());
  assertEquals(null, book1.getBorrower());
}
```

9-15
```
public boolean isAvailable() {
  return borrower == null;
}
```

9-16 Fill in the blanks in this test method to verify `getTitle` works so all assertions pass.
```
assertTrue(book1.isAvailable());
assertTrue (book2.isAvailable());
book1.borrowBook("Sam Mac");
book2.borrowBook("Sam Mac");
assertFalse(book1.isAvailable());
assertFalse(book2.isAvailable());
assertEquals("_Sam _", book1.getBorrower());
assertEquals("_Li__", book2.getBorrower());
```

9-17
```
public String toString() {
  return "" + ID + " " + balance;
}
```

Chapter 10

An Array Instance Variable

Goal

- Implement a type that uses an array instance variable.

10.1 `StringBag` — A Simple Collection Class

As you continue your study of computing fundamentals, you will spend a fair amount of time using arrays and managing collections of data. The Java array is one of several data storage structures used inside classes with the main task of storing a collection. These are known as collection classes with some of the following characteristics:

- The main responsibility of a collection class is to store a collection of objects
- Objects are added and removed from a collection
- A collection class allows clients to access the individual elements
- A collection class may have search-and-sort operations for locating a particular item.
- Some collections allow duplicate elements; other collections do not

The Java array uses subscript notation to access individual elements. The collection class shown next exemplifies a higher-level approach to storing a collection of objects. It presents users with messages and hides the array processing details inside the methods. The relatively simple collection class also provides a review of Java classes and methods. This time, however, the class will have an array instance variable. The methods will employ array-processing algorithms. More specifically, this collection will represent a bag. Bag is a mathematical term for an unordered collection of values that may have duplicates. It is also know as a multi-set. This bag will be store a collection of strings and will be named `StringBag`. A `StringBag` object will have the following characteristics:

- A `StringBag` object can store a collection of `String` objects
- `StringBag` elements need not be unique, duplicates are allowed
- The order of elements is not important
- Programmers can ask how many occurrences of a String are in the bag (may be 0)
- Elements can be removed from a `StringBag` object
- This `StringBag` class is useful for learning about collections, array processing, Java classes and Test-Driven Development.

A `StringBag` object can store any number of `String` objects. A `StringBag` object will understand the messages such as `add`, `remove` and `occurencesOf`. The design of `StringBag` is provided here as three commented method headings.

```
// Put stringToAdd into this StringBag (order not important)
public void add(String stringToAdd);

// Return how often element equals an element in this StringBag
public int occurencesOf(String element);

// Remove one  occurrence of stringToRemove if found and return true.
// Return false if stringToRemove is not found in this StringBag.
public boolean remove(String stringToRemove);
```

Using Test Driven Development, the tests come first. Which method should be tested first? It's difficult to implement only one and know it works. If we work on `add` alone, how do we know an element has actually been added. One solution is to develop `occurencesOf` at the same time and verify both are working together. A test method could add several elements and verify they are there with `occurencesOf`. We should also verify contains returns false for elements in the bag. So `add(String)` and `occurencesOf(String)` will be developed first. We'll begin with a unit test with one test method that adds one element. `occurencesOf` should return 0 before add and 1 after.

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class StringBagTest {

  @Test
  public void testAddAndOccurencesOfForOnlyOneElement () {
    StringBag friends = new StringBag();
    friends.add("Sage");
    assertEquals(1, friends.occurencesOf("Sage"));
  }
}
```

Of course, this unit test will not compile. The class doesn't even exist; nor do the `add` and `occurencesOf` methods; nor does the constructor. The following start at a `StringBag` type at least allows the unit test to compile. The assertions will not pass, at least not yet. All methods are written as stubs—a temporary substitute for yet-to-be-developed code.

```
// A class for storing a multi-set (bag) of String elements.
public class StringBag {

  // Construct an empty StringBag object (no elements stored yet)
  public StringBag() {
    // TODO Complete this method
  }

  // Add an element to this StringBag
  public void add(String stringToAdd) {
     // TODO Complete this method
  }

  // Return how often element equals an element in this StringBag
  public int occurencesOf(String element) {
     // TODO Complete this method
     return 0;
  }
}
```

The **StringBag** Constructor

The private instance variables of the `StringBag` class include an array named `data` for storing a collection of `String` objects. Each `StringBag` object also has an integer named `n` to maintain the number of meaningful elements that are in the `StringBag`. The `add` and `occurencesOf` methods will need both instance variables to accomplish their responsibilities. The constructor establishes an empty `StringBag` object by setting `n` to zero. The array capacity is set to the arbitrary initial capacity of 10. We don't know how big the collection will grow to when used *later—we* will deal with that later.

```
public class StringBag {

  private String[] data; // Stores the collection
  private int n;         // Current number of elements

  // Construct an empty StringBag object
  public StringBag() {
    n = 0;
    data = new String[10]; // Initial capacity is 10
  }
```

`public void add(String stringToAdd)`

Both `n` and `data` must be available to the `add` method. This is not a problem, since any `StringBag` method has access to the private instance variables of `StringBag`. To add an element to the `StringBag`, the argument reference passed to the `stringToAdd` parameter can be placed at the "end" of the array, or more specifically, at the first available array location. This two-step algorithm summarizes how a new `String` is added to the first available array position:

***Algorithm:* Adding an element**

`data[n]` = *the-argument-passed-to-`StringBag.add`*
increment `n` by `+1`

The argument passed to `StringBag`'s `add` method is stored into the proper array location using `n` as the index. Then `n` gets incremented by 1 to reflect the new addition. Incrementing `n` by 1 maintains the number of elements in the `StringBag`.

Incrementing `n` also conveniently sets up a situation where the next added element is inserted into the proper array location. The array location at `data[n]` is the next place to store the next element can be placed. This is demonstrated in the following view of the state of the `StringBag` before and after the string `"and a fourth"` after this code executes

```
StringBag bag = new StringBag();
bag.add("A string");
bag.add("Another string");
bag.add("and still another");
```

Before		After	
Instance Variables	**State of bagOfStrings**	**Instance Variable**	**State of bagOfStrings**
data[0]	"A string"	data[0]	"A string"
data[1]	"Another string"	data[1]	"Another string"
data[2]	"and still another"	data[2]	"and still another"
data[3]	null // next available	**data[3]**	**"and a fourth"**
data[4]	null	data[4]	null // next available
...	...	...	...
data[9]	null	data[9]	null
n	3	n	4

Here is the `add` method that places new elements at the first available location. It is important to keep the elements together. Don't allow null between elements. This method ensures nulls are not in the mix.

```
// Add an element to this StringBag
public void add(String stringToAdd) {
  // Store the reference into the array
  data[n] = stringToAdd;
  // Make sure n is always increased by one
  n++;
}
```

The unit test is run, but the single test method does not pass; `occurencesOf` still does nothing.

`public int occurencesOf(String element)`

Since there is no specified ordering for Bags in general or `StringBag` in particular, the element passed as an argument may be located at any index. Also, a value that equals the argument may occur more than once. Thus each element in indexes 0..n-1 must be compared. It makes the most sense to use the `equals` method, assuming `equals` has been overridden to compare the state of two objects rather than the reference values. And with `String`, equals does compare state.

By setting `result` to 0 below, the `occurencesOf` method first states there are no elements equal to `element`.

```
// Return how often element equals an element in this StringBag
public int occurencesOf(String element) {
  int result = 0;
  for (int subscript = 0; subscript < n; subscript++) {
    if (element.equals(data[subscript]))
      result++;
  }
  return result;
}
```

The for loop then iterates over every meaningful element in the array. Each time `element` equals any array element, `result` increments by 1. Our first assertion passes.

```
@Test
public void testAddAndOccurencesOfForOnlyOneElement() {
  StringBag friends = new StringBag();
  friends.add("Sage");
  assertEquals(1, friends.occurencesOf("Sage"));
}
```

Other Test Methods

Another test method verifies that duplicate elements are can exist and are found.

```
@Test
public void testOccurencesOf() {
  StringBag names = new StringBag();
  names.add("Tyler");
  names.add("Devon");
  names.add("Tyler");
  names.add("Tyler");
  assertEquals(1, names.occurencesOf("Devon"));
  assertEquals(3, names.occurencesOf("Tyler"));
}
```

Another test method verifies 0 is returned when the String argument is not in the bag.

```
@Test
public void testOccurencesOfWhenItShyouldReturnZeros() {
  StringBag names = new StringBag();
  assertEquals(0, names.occurencesOf("Devon"));
  assertEquals(0, names.occurencesOf("Tyler"));
  names.add("Sage");
  names.add("Hayden");
  assertEquals(0, names.occurencesOf("Devon"));
  assertEquals(0, names.occurencesOf("Tyler"));
}
```

Another test method documents that this collection is case sensitive.

```
@Test
public void testOccurencesOfForCaseSensitivity() {
  StringBag names = new StringBag();
  names.add("UPPER");
  names.add("Lower");

  // Not in the bag (case sensitive)
  assertEquals(0, names.occurencesOf("upper"));
  assertEquals(0, names.occurencesOf("lower"));

  // In the bag
  assertEquals(1, names.occurencesOf("UPPER"));
  assertEquals(1, names.occurencesOf("Lower"));
}
```

Yet another test method tries to add 500 strings only to find something goes wrong.

```
@Test
public void testAdding500Elements() {
  StringBag bag = new StringBag();
  for (int count = 1; count <= 500; count++) {
    bag.add("Str#" + count);
  }
  assertEquals(1, bag.occurencesOf("Str#1"));
  assertEquals(1, bag.occurencesOf("Str#2"));
  assertEquals(1, bag.occurencesOf("Str#499"));
  assertEquals(1, bag.occurencesOf("Str#500"));
}
java.lang.ArrayIndexOutOfBoundsException: 10
at StringBag.add(StringBag.java:34)
at StringBagTest.testAdding500Elements(StringBagTest.java:39)
```

After 10 adds, n == 10. The attempt to store the 11th element in the StringbBag results in an `ArrayIndexOutOfBounds` exception with the attempt to assign an element to `data[10]`.

Before any new `String` is added, a check should be made to ensure that there is the capacity to add another element. If the array is filled to capacity (`n == data.length`) there is not enough room to add the new element. In this case, we need to increase the array capacity.

The code to increase the capacity of the array could be included in the `add` method. However this task is complex enough that it will be placed into a "helper" method named `growArray`. The `add` method changes with a guarded action: grow the array only when necessary.

```
public void add(String stringToAdd) {
  // Make sure the array can store a new element
  if (n == data.length) {
    growArray();
  }
  // Store the reference into the array
  data[n] = stringToAdd;
  // Make sure my_size is always increased by one
  n++;
}
```

The `growArray` method will help this `add` method perform its task with less code. The `add` method delegates a well-defined responsibility of growing the array to another method. This makes for more readable and maintainable code.

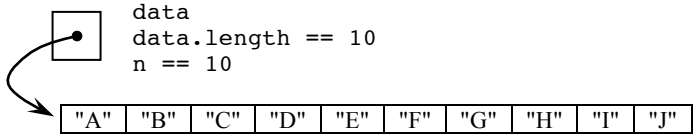
`private void growArray()`

Because `growArray` is inside class `StringBag`, any `StringBag` object can send a `growArray` message to itself. The message was sent from this object in `add`. And because `data` is an instance variable, any `StringBag` object can change `data` to reference a new array with more capacity. This is done with the following algorithm:

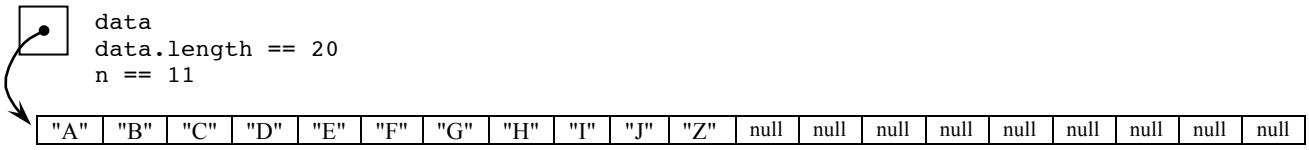
- Make a temporary array that is bigger (by 10) than the instance variable.
- Copy the original contents (`data[0]` through `data[n - 1]`) into this temporary array.
- Assign the reference to the temporary array to the array instance variable

```
// Change data to have the same elements in indexes 0..n - 1
// and have the same number of new array locations to store new elements.
private void growArray() {
  String[] temp = new String[n + 10];
  // Copy all existing elements into the new and larger array
  for (int index = 0; index < n; index++) {
    temp[index] = data[index];
  }
  // Store a reference to the new bigger array as part of this object's state
  data = temp;
}
```

When the array is filled to capacity (with the Strings `"A"` through `"J"` added in this example), the instance variables `data` and `n` look like this:



During the message `add("Z");`, the `add` method would send the `growArray` message in order to increase the capacity by 10. The instance variables would change to this picture of memory:



Note: The `growArray` method is declare private because it is better design to *not* clutter the public part of a class with things that users of the class are not able to use or are not interested in using. It is good practice to hide details from users of your software.

`public boolean remove(String stringToRemove)`

If `stringToRemove` is found to equal one of the strings referenced by the array, `remove` effectively takes one of the occurrences of the String element. Consider the following test method that attempts to remove `"Not in the bag"`.

```
@Test
public void testRemoveOneThatIsThereAnotherThatIsNot() {
  StringBag bag = new StringBag();
  bag.add("A string");
  bag.add("Another string");
  bag.add("and still another");
  bag.add("and a fourth");
  assertFalse(bag.remove("Not in the bag"));
  assertTrue(bag.remove("Another string"));
}
```

Here are the values of the instance variables `data` and `n` and of the local objects `index` and `stringToRemove` while trying to remove `"Another string"`:

Instance Variable	State of bag
data[0]	"A string"
data[1]	"Another string"
data[2]	"and still another"
data[3]	"and a fourth"
data[4]	null
...	...
data[9]	null
n	4

The algorithm used to remove an element is in these steps (other algorithms also work).

- Find the index of an element to remove, or set to -1 if `stringToRemove` does not exist
- If the index != -1, move the element at the end of the array to this index
- Decrement n (n--)

The remove algorithm calls the private helper method `indexOf` that has the purpose of returning an index of the string to be removed. If the string does not equal an array element, the `indexOf` method (discussed later) returns -1. In this case of trying to remove the string `"Not in the bag"` the method simply returns false. The method terminated and the first assertion (above) passes.

```
// Remove an element that equals stringToRemove if found and return true.
// Return false if stringToRemove was not found in this StringBag.
public boolean remove(String stringToRemove) {
  // indexOf returns the index of an element that equals stringToRemove
  // or -1 if stringToRemove is not in this bag.
  int subscript = indexOf(stringToRemove);
  if (subscript == -1)
    return false;
  else {   // . . .
```

In the 2nd assertion `assertTrue(bag.remove("Another string"));` that attempts to remove an element that does exist, the array will be changed, `n` will be changed, and `indexOf` will return true. These variables that are local to `remove` indicate the string was found at index 1.

Local Variable	State of `remove`'s Local Variable after a Sequential Search
`stringToRemove`	`"Another string"`
`index`	`1`

Once found, the reference stored in `data[index]` must somehow be removed from the array, which is currently `data[1]` or `"Another string"`. The simple way to do this is to move the last element into the spot where `stringToRemove` was found. It is okay to destroy the reference in `data[1]`. This is the object to be removed from the `StringBag`. Also, since there is no ordering requirement, it is also okay to move `data[n - 1]`, which is the last meaningful element in the array. When `n--` occurs, the 2nd reference to the string at data[n-1] is no longer considered to be in the collection. Although not necessary, this code assigns `null` to that 2nd unneeded reference.

```
      // Move the last string in the array to where stringToRemove was found.
      data[subscript] = data[n - 1];
      // Mark old array element as no longer holding a reference (not required)
      data[n - 1] = null;
      // Decrease this StringBag's number of elements
      n--;

      // Let this method return true to where the message was sent
      return true;
    }
  } // End method remove
```

The state of `StringBag` now looks like this (three changes are highlighted):

Instance Variable	State of `bagOfStrings`	
`data[0]`	`"A string"`	
`data[1]`	`"And a fourth"`	Overwrite `"another string"`
`data[2]`	`"and still another"`	
`data[3]`	`null`	`data[3]` is no longer meaningful
`data[4]`	`null`	
...		
`data[9]`	`null`	
`n`	`3`	`n` is 3 now

Although the elements are not in the same order (this was not a requirement), the same elements exist after the requested removal. Because the last element has been relocated, `n` must decrement by 1. There are now only three, not four, elements in this `StringBag` object.

The same code works even when removing the last element. The assignment is done. Decreasing `n` by one effectively eliminates the last element.

`private int indexOf(String element)`

The remove method used another method to find the index of an element to remove (or -1 if no element found). Although this code could have gone in remove, the well-defined responsibility of finding the index of an element in an array was placed in this private helper method to keep the remove algorithm a bit simpler. The `indexOf` method will sequentially search each array element beginning at index 0 until one of two things happen.

1. element equals an array element and that index of that element is returned to method `remove(String element)`

2. the loop terminates because there are no more element to examine. In this case, `indexOf` returns -1 to method `remove(String element)`

```
// Return the index of the first occurrence of stringToRemove.
private int indexOf(String element) {
  // Look at all elements until the string
  for (int index = 0; index < n; index++) {
    if (element.equals(data[index]))
      return index;
  }
  // Otherwise result is not changed from -1.
  return -1;
}
```

Again we see a helper method declared `private` because `indexOf` is currently considered a method that programmers are *not* meant to use. It was not in the specification. Here is the complete `StringBag` class.

```
// A class for storing an unordered collection of Strings.
// This class was designed to provide practice and review in
// implementing methods and classes along with using arrays.
public class StringBag {

  private String[] data; // Stores the collection
  private int n; // Current number of elements

  // Construct an empty StringBag object
  public StringBag() {
    n = 0;
    data = new String[10]; // Initial capacity is 10
  }


  // Return the element at the specified index.
  // Precondition: index >= 0 && index < size()
  public String get(int index) {
    return data[index];
  }

  // Add a string to the StringBag in no particular place.
  // Always add StringToAdd (unless the computer runs out of memory)
  public void add(String stringToAdd) {
    // Make sure the array can store a new element
    if (n == data.length) {
      growArray();
    }
    // Store the reference into the array
    data[n] = stringToAdd;
    // Make sure my_size is always increased by one
    n++;
  }

  // Change data to have the same elements in indexes 0..n - 1 and have
  // the same number of new array locations to store new elements.
  private void growArray() {
    String[] temp = new String[n + 10];
    // Copy all existing elements into the new and larger array
    for (int index = 0; index < n; index++) {
      temp[index] = data[index];
    }
    // Store a reference to the new bigger
    // array as part of this object's state
    data = temp;
  }
```

```
  // Return how often element equals an element in this StringBag
  public int occurencesOf(String element) {
    int result = 0;
    for (int subscript = 0; subscript < n; subscript++) {
      if (element.equals(data[subscript]))
        result++;
    }
    return result;
  }

  // Remove an element that equals stringToRemove if found and return true.
  // Return false if stringToRemove was not found in this StringBag.
  public boolean remove(String stringToRemove) {
    int subscript = indexOf(stringToRemove);
    if (subscript == -1)
      return false;
    else {
      // Move the last string in the array to where stringToRemove was found.
      data[subscript] = data[n - 1];
      // Mark old array element as no longer holding a reference (not required)
      data[n - 1] = null;
      // Decrease this StringBag's number of elements
      n--;
      return true;
    }
  }

  // Return the index of the first occurrence of stringToRemove.
  // Otherwise return -1 if stringToRemove is not found.
  private int indexOf(String element) {
    // Look at all elements until the string
    for (int index = 0; index < n; index++) {
      if (element.equals(data[index]))
        return index;
    }
    // Otherwise result is not changed from -1.
    return -1;
  }
}  // End class StringBag
```

Other Test Methods

The `remove` method and its `indexOf` method are complex. Further testing is appropriate. This test verifies that all duplicates can be removed.

```
  @Test
  public void testRemoveWhenDuplicatedO() {
    StringBag bag = new StringBag();
    bag.add("A");
    bag.add("B");
    bag.add("B");
    bag.add("B");
    bag.add("A");

    assertEquals(3, bag.occurencesOf("B"));
    assertTrue(bag.remove("B"));
    assertEquals(2, bag.occurencesOf("B"));

    assertTrue(bag.remove("B"));
    assertEquals(1, bag.occurencesOf("B"));
```

```
    assertTrue(bag.remove("B"));
    assertEquals(0, bag.occurencesOf("B"));

    // There should be no more Bs
    assertFalse(bag.remove("B"));
    assertEquals(0, bag.occurencesOf("lower"));
  }
```

Other tests should be made for these situations:

- when the bag is empty
- when there is one element, try removing an element that is not there
- when there is one element, try removing an element that *is* there
- remove all elements when size > 2

```
  @Test
  public void testRemoveWhenEmpty() {
    StringBag bag = new StringBag();
    assertEquals(0, bag.occurencesOf("B"));
    assertFalse(bag.remove("Not here"));
    assertEquals(0, bag.occurencesOf("B"));
  }

  @Test
  public void testRemoveNonExistentElementWhenSizeIsOne() {
    StringBag bag = new StringBag();
    bag.add("Only one element");
    assertEquals(1, bag.occurencesOf("Only one element"));
    assertFalse(bag.remove("Not here"));
    assertEquals(1, bag.occurencesOf("Only one element"));
  }

  @Test
  public void testRemoveElementWhenSizeIsOne() {
    StringBag bag = new StringBag();
    bag.add("Only one element");
    assertEquals(1, bag.occurencesOf("Only one element"));
    assertTrue(bag.remove("Only one element"));
    assertEquals(0, bag.occurencesOf("Only one element"));
  }

  @Test
  public void testRemoveAllElementsWhenSizeGreaterThanTwo() {
    StringBag bag = new StringBag();
    bag.add("A");
    bag.add("B");
    bag.add("C");
    assertTrue(bag.remove("A"));
    assertTrue(bag.remove("B"));
    assertTrue(bag.remove("C"));
    assertEquals(0, bag.occurencesOf("A"));
    assertEquals(0, bag.occurencesOf("B"));
    assertEquals(0, bag.occurencesOf("C"));
  }
```

Self-Check

10-1 What happens when an attempt is made to remove an element that is not in the bag.

10-2 Using the implementation of `remove` just given, what happens when an attempt is made to remove an element from an empty `StringBag` (`n == 0`)?

10-3 Must `remove` always maintain the `StringBag` elements in the same order as that in which they were originally added?

10-4 What happens when an attempt is made to remove an element that has two of the same values in the `StringBag`?

10-5 Write the output of the following code:

```
StringBag aBag = new StringBag();
aBag.add("First");
aBag.add("Second");
aBag.add("Third");
System.out.println(aBag.occurencesOf("first"));
System.out.println(aBag.occurencesOf("Second"));
System.out.println(aBag.remove("First"));
System.out.println(aBag.remove("Third"));
System.out.println(aBag.remove("Third"));
System.out.println(aBag.occurencesOf("first"));
System.out.println(aBag.occurencesOf("Second"));
```

Answers to Self-Checks

10-1 `remove` returns `false`, the `StringBag` object does not change.

10-2 Nothing noticeable to the user happens. The loop test (`index < my_size`) is false immediately, so index remains 0. Then the expression `if( index == my_size )` is true and false is returned.

10-3 No. The last element may be moved to the first vector position, or the second, or anywhere else. There are other collections used to store elements in order.

10-4 `StringBag remove` removes the first occurrence. All other occurrences of the same value remain in the bag.

10-5

```
0
1
true
true
false
0
1
```

Chapter 11

Two-Dimensional Arrays

This chapter introduces Java arrays with two subscripts for managing data logically stored in a table-like format—in rows and columns. This structure proves useful for storing and managing data in many applications, such as electronic spreadsheets, games, topographical maps, and student record books.

11.1 2-D Arrays

Data that conveniently presents itself in tabular format can be represented using an array with two subscripts, known as a two-dimensional array. Two-dimensional arrays are constructed with two pairs of square brackets to indicate two subscripts representing the row and column of the element.

General Form: **A two-dimensional array construction (all elements set to default values)**

type **[ ] [ ]** *array-name* **= new** *type* **[** *row-capacity* **] [** *column-capacity* **];**
type **[ ] [ ]** *array-name* **= {** { element[0][0], element[0][1], element[0][2], … } **,**
{ element[1][0], element[1][1], element[1][2], … } **,**
{ element[2][0], element[2][1], element[2][2], … } **};**

- *type* may be one of the primitive types or the name of any Java class or interface
- *identifier* is the name of the two-dimensional array
- *rows* specifies the total number of rows
- *columns* specifies the total number of columns

Examples:

```
double[][] matrix = new double[4][8];

// Construct with integer expressions
int rows = 5;
int columns = 10;
String[][] name = new String[rows][columns];

// You can use this shortcut that initializes all elements
int[][] t = { { 1, 2, 3 },   // First row of 3 integers
              { 4, 5, 6 },   // Row index 1 with 3 columns
              { 7, 8, 9 } }; // Row index 2 with 3 columns
```

Referencing Individual Items with Two Subscripts

A reference to an individual element of a two-dimensional array requires two subscripts. By convention, programmers use the first subscript for the rows, and the second for the columns. Each subscript must be bracketed individually.

***General Form:* Accessing individual two-dimensional array elements**

two-dimensional-array-name **[** *rows* **] [** *columns* **]**

- *rows* is an integer value in the range of 0 through the number of rows - 1
- *columns* is an integer value in the range of 0 through the number of columns - 1

Examples:

```
String[][] name = new String[5][10];
name[0][0] = "Upper Left";
name[4][9] = "Lower Right";
assertEquals("Upper Left", name[0][0]);

// name.length is the number of rows,
// name[0].length is the number of columns
assertEquals("Lower Right", name[name.length-1][name[0].length-1]);
```

Nested Looping with Two-Dimensional Arrays

Nested looping is commonly used to process the elements of two-dimensional arrays. This initialization allocates enough memory to store 40 floating-point numbers—a two-dimensional array with five rows and eight columns. Java initializes all values to 0.0 when constructed.

```
int ROWS = 5;
int COLUMNS = 8;
double[][] table = new double[ROWS][COLUMNS];  // 40 elements set to 0.0
```

These nested `for` loops initialize all 40 elements to `-1.0`.

```
// Initialize all elements to -1.0
for (int row = 0; row < ROWS; row++) {
  for (int col = 0; col < COLUMNS; col++) {
    table[row][col] = -1.0;
  }
}
```

Self-Check

Use this construction of a 2-D array object to answer questions 1 through 8:

```
int[][] a = new int[3][4];
```

11-1 What is the value of `a[1][2]`?

11-2 Does Java check the range of the subscripts when referencing the elements of `a`?

11-3 How many `int`s are properly stored by `a`?

11-4 What is the row (first) subscript range for `a`?

11-5 What is the column (second) subscript range for `a`?

11-6 Write code to initialize all of the elements of `a` to 999.

11-7 Declare a two-dimensional array `sales` to store 120 floating-point numbers in 10 rows.

11-8 Declare a two-dimensional array named `sales2` such that 120 floating-point numbers can be stored in 10 columns.

A two-dimensional array manages tabular data that is typically processed by row, by column, or in totality. These forms of processing are examined in an example class that manages a grade book. The data could look like this with six quizzes for each of the nine students.

```
   Quiz #0        1        2        3        4        5
0     67.8     56.4     88.4     79.1     90.0     66.0
1     76.4     81.1     72.2     76.0     85.6     85.0
2     87.8     76.4     88.7     83.0     76.3     87.0
3     86.4     54.0     40.0      3.0      2.0      1.0
4     72.8     89.0     55.0     62.0     68.0     77.7
5     94.4     63.0     92.9     45.0     75.6     99.5
6     85.8     95.0     88.1    100.0     60.0     85.8
7     76.4     84.4    100.0     94.3     75.6     74.0
8     57.9     49.5     58.8     67.4     80.0     56.0
```

This data will be stored in a tabular form as a 2D array. The 2D array will be processed in three ways:

1. Find the average quiz score for any of the 9 students
2. Find the range of quiz scores for any of the 5 quizzes
3. Find the overall average of all quiz scores

Here are the methods that will be tested and implemented on the next few pages:

```
// Return the number of students in the data (#rows)
public int getNumberOfStudents()

// Return the number of quizzes in the data (#columns)
public int getNumberOfQuizzes()

// Return the average quiz score for any student
public double studentAverage(int row)

// Return the range of any quiz
public double quizRange(int column)

// Return the average of all quizzes
public double overallAverage()
```

Reading Input from a Text File

In programs that require little data, interactive input suffices. However, initialization of arrays quite often involves large amounts of data. The input would have to be typed in from the keyboard many times during implementation and testing. That much interactive input would be tedious and error-prone. So here we will be read the data from an external file instead.

The first line in a valid input file specifies the number of rows and columns of the input file. Each remaining line represents the quiz scores of one student.

```
9   6
67.8   56.4   88.4   79.1   90.0   66.0
76.4   81.1   72.2   76.0   85.6   85.0
87.8   76.4   88.7   83.0   76.3   87.0
86.4   54.0   40.0    3.0    2.0    1.0
72.8   89.0   55.0   62.0   68.0   77.7
94.4   63.0   92.9   45.0   75.6   99.5
85.8   95.0   88.1  100.0   60.0   85.8
76.4   84.4  100.0   94.3   75.6   74.0
57.9   49.5   58.8   67.4   80.0   56.0
```

The first two methods to test will be the two getters that determine the dimensions of the data. The actual file used in the test has 3 students and 4 quizzes. The name of the file will be passed to the `QuizData` constructor.

```
@Test
public void testGetters() {
  /* Process this small file that has 3 students and 4 quizzes.
   3 4
   0.0 10.0 20.0 30.0
   40.0 50.0 60.0 70.0
   80.0 90.0 95.5 50.5
  */
  QuizData quizzes = new QuizData("quiz3by4");
  assertEquals(3, quizzes.getNumberOfStudents());
  assertEquals(4, quizzes.getNumberOfQuizzes());
}
```

The name of the file will be passed to the `QuizData` constructor that then reads this text data using the familiar Scanner class. However, this time a new File object will be needed. And this requires some understanding of exception handling.

Exception Handling when a File is Not Found

When programs run, errors occur. Perhaps an arithmetic expression results in division by zero, or an array subscript is out of bounds, or there is an attempt to read a file from a disk using a specific file name that does not exist. Or perhaps, the expression in an array subscript is negative or 1 greater than the capacity of that array. Programmers have at least two options for dealing with these types of exception:

- Ignore the exception and let the program terminate
- Handle the exception

However, in order to read from an input file, you cannot ignore the exception. Java forces you to try to handle the exceptional event. Here is the code the tries to have a Scanner object read from an input file named quiz.data. Notice the argument is now a new File object.

```
Scanner inFile = new Scanner(new File("quiz.data));
```

This will not compile. Since the file "quiz.data" may not be found at runtime, the code may throw a `FileNotFoundException`. In this type of exception (called a checked exception), Java requires that you put the construction in a `try` block—the keyword `try` followed by the code wrapped in a block, `{ }`.

```
try {
    code that may throw an exception when an exception is thrown
}
catch (Exception anException) {
    code that executes only if an exception is thrown from code in the above try block.
}
```

Every `try` block must be followed by a at least one `catch` block—the keyword `catch` followed by the anticipated exception as a parameter and code wrapped in a block. The `catch` block contains the code that executes when the code in the `try` block causes an exception to be thrown (or called a method that throws an exception). So to get a Scanner object to try to read from an input file, you need this code.

```
    Scanner inFile = null;
    try {
      inFile = new Scanner(new File(fileName));
    }
    catch (FileNotFoundException fnfe) {
      System.out.println("The file '" + fileName + " was not found");
    }
```

This will go into the `QuizData` constructor that reads the first two integers as the number of rows followed by the number of columns as integers. The file it reads from is passed as a string to the constructor. This allows the programmer to process data stored in a file (assuming the data is properly formatted and has the correct amount of input.

```
// A QuizData object will read data from an input file an allow access to
// any students quiz average, the range of any quiz, and the average quiz
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class QuizData {
  // Instance variables
  private double[][] quiz;
  private int numberOfStudents;
  private int numberOfQuizzes;

  public QuizData(String fileName) {
    Scanner inFile = null;
    try {
      inFile = new Scanner(new File(fileName));
    }
    catch (FileNotFoundException e) {
      System.out.println("The file '" + fileName + " was not found");
    }
   }  // More to come ...
```

Because the private instance variables members are known throughout the `QuizData` class, the two-dimensional array named `quiz` can, from this point forward, communicate its subscript ranges for both rows and columns at any time and in any method. These values are stored

```
  // Get the dimensions of the array from the input file
  numberOfStudents = inFile.nextInt();
  numberOfQuizzes = inFile.nextInt();
```

The next step is to allocate memory for the two-dimensional array:

```
   quiz = new double[numberOfStudents][numberOfQuizzes];
```

Now with a two-dimensional array precisely large enough to store `numberOfStudents` rows of data with `numberOfQuizzes` quiz scores in each row, the two-dimensional array gets initialized with the file data using nested `for` loops.

```
   // Initialize a numberOfStudents-by-numberOfQuizzes array
   for (int row = 0; row < getNumberOfStudents(); row++) {
     for (int col = 0; col < getNumberOfQuizzes(); col++) {
       quiz[row][col] = inFile.nextDouble();
     }
   }
 } // End QuizData(String) constructor
```

`QuizData` also has these getters now s the first test method has both assertions passing

```
public int getNumberOfStudents() {
  return numberOfStudents;
}

public int getNumberOfQuizzes() {
  return numberOfQuizzes;
}
```

However, more tests are required to verify the 2D array is being initialized properly. One way to do this is to have a `toString` method so the array can be printed.

Self-Check

11-9 Write method `toString` to print all elements of any `QuizData` object to look like this:

```
0.0 10.0 20.0 30.0
40.0 50.0 60.0 70.0
80.0 90.0 95.5 50.5
```

Student Statistics: Row by Row Processing

To further verify the array was initialized, we can write a test to make sure all three students have the correct quiz average.

```
@Test
public void testStudentAverage() {
/* Assume the text file "quiz3by4" has these four lines of input data:
3 4
0.0 10.0 20.0 30.0
40.0 50.0 60.0 70.0
80.0 90.0 95.5 50.5
*/
  QuizData quizzes = new QuizData("quiz3by4");
  assertEquals(15.0, quizzes.studentAverage(0), 0.1);
  assertEquals(220.0 / 4, quizzes.studentAverage(1), 0.1);
  assertEquals((80.0+90.0+95.5+50.5) / 4, quizzes.studentAverage(2), 0.1);
}
```

The average for one student is found by adding all of the elements of one row and dividing by the number of quizzes. The solution uses the same row as `col` changes from 0 through 3.

```
// Return the average quiz score for any student
public double studentAverage(int row) {
  double sum = 0.0;
  for (int col = 0; col < getNumberOfQuizzes(); col++) {
    sum = sum + quiz[row][col];
  }
  return sum / getNumberOfQuizzes();
}
```

Quiz Statistics: Column by Column Processing

To even further verify the array was initialized, we can write a test to ensure correct quiz ranges.

```
@Test
public void testQuizAverage() { // Assume the file "quiz3by4" has these lines
  // 3 4
  // 0.0 10.0 20.0 30.0
  // 40.0 50.0 60.0 70.0
  // 80.0 90.0 95.5 50.5
```

```
    QuizData quizzes = new QuizData("quiz3by4");
    assertEquals(80.0, quizzes.quizRange(0), 0.1);
    assertEquals(80.0, quizzes.quizRange(1), 0.1);
    assertEquals(75.5, quizzes.quizRange(2), 0.1);
    assertEquals(40.0, quizzes.quizRange(3), 0.1);
  }
```

The range for each quiz is found by first initializing the min and the max by the quiz score in the given column. The loop uses the same column as `row` changes from 1 through 3 (already checked row 0). Inside the loop, the current value is compared to both the min and the max to ensure the max – min is the correct range.

```
  // Find the range for any given quiz
  public double quizRange(int column) {
    // Initialize min and max to the first quiz in the first row
    double min = quiz[0][column];
    double max = quiz[0][column];
    for (int row = 1; row < getNumberOfStudents(); row++) {
      double current = quiz[row][column];
      if (current < min)
        min = current;
      if (current > max)
        max = current;
    }
    return max - min;
  }
```

Overall Quiz Average: Processing All Rows and Columns

The test for overall average shows that an expected value of 49.67.

```
  @Test
  public void testOverallAverage() {
    QuizData quizzes = new QuizData("quiz3by4");
    assertEquals(49.7, quizzes.overallAverage(), 0.1);
  }
```

Finding the overall average is a simple matter of summing every single element in the two-dimensional array and dividing by the total number of quizzes.

```
  public double overallAverage() {
    double sum = 0.0;
    for (int studentNum = 0; studentNum < getNumberOfStudents(); studentNum++) {
      for (int quizNum = 0; quizNum < getNumberOfQuizzes(); quizNum++) {
        sum += quiz[studentNum][quizNum];
      }
    }
    return sum / (getNumberOfQuizzes() * getNumberOfStudents());
  }
```

Answers to Self-Checks

11-1 0.0

11-2 Yes

11-3 12

11-4 0 through 2 inclusive

11-5 0 through 3 inclusive

11-6
```
for (int row = 0; row < 3; row++) {
  for (int col = 0; col < 4; col++) {
    a [row][col] = 999;
  }
}
```

11-7
```
double[][]sales = new double[10][12];
```

11-8
```
double[][]sales2 = new double[12][10];
```

11-9
```
public String toString() {
  String result = "";
  for (int studentNum = 0; studentNum < getNumberOfStudents(); studentNum++){
    for (int quizNum = 0; quizNum < getNumberOfQuizzes(); quizNum++) {
      result += " " + quiz[studentNum][quizNum];
    }
    result += "\n";
  }
  return result;
}
```

Chapter 12

Algorithm Analysis

Goals

- Analyze algorithms
- Understand some classic searching and sorting algorithms
- Distinguish runtimes that are O(1), O(log n), O(n), O(n log n), and O(n^2)

12.1 Algorithm Analysis

This chapter introduces a way to investigate the efficiency of algorithms. Examples include searching for an element in an array and sorting elements in an array. The ability to determine the efficiency of algorithms allows programmers to better compare them. This helps when choosing a more efficient algorithm when implementing data structures.

An **algorithm** is a set of instructions that can be executed in a finite amount of time to perform some task. Several properties may be considered to determine if one algorithm is better than another. These include the amount of memory needed, ease of implementation, robustness (the ability to properly handle exceptional events), and the relative efficiency of the runtime.

The characteristics of algorithms discussed in this chapter relate to the number of operations required to complete an algorithm. A tool will be introduced for measuring anticipated runtimes to allow comparisons. Since there is usually more than one algorithm to choose from, these tools help programmers answers the question: "Which algorithm can accomplish the task more efficiently?"

Computer scientists often focus on problems related to the efficiency of an algorithm: Does the algorithm accomplish the task fast enough? What happens when the number of elements in the collection grows from one thousand to one million? Is there an algorithm that works better for storing a collection that is searched frequently? There may be two different algorithms that accomplish the same task, but all other things being equal, one algorithm may take much longer than another when implemented and run on a computer.

Runtimes may be reported in terms of actual time to run on a particular computer. For example, SortAlgorithmOne may require 2.3 seconds to sort 2000 elements while SortAlgorithmTwo requires 5.7 seconds. However, this time comparison does not ensure that SortAlgorithmOne is better than SortAlgorithmTwo. There could be a good implementation of one algorithm and a poor implementation of the other. Or, one computer might have a special hardware feature that SortAlgorithmOne takes advantage of, and without this feature SortAlgorithmOne would not be faster than SortAlgorithmTwo. Thus the goal is to compare algorithms, not programs. By comparing the actual running times of SortAlgorithmOne and SortAlgorithmTwo, programs are being considered—not their algorithms. Nonetheless, it can

prove useful to observe the behavior of algorithms by comparing actual runtimes — the amount of time required to perform some operation on a computer. The same tasks accomplished by different algorithms can be shown to differ dramatically, even on very fast computers. Determining how long an algorithm takes to complete is known as algorithm analysis.

Generally, the larger the size of the problem, the longer it takes the algorithm to complete. For example, searching through 100,000 elements requires more operations than searching through 1,000 elements. In the following discussion, the variable **n** will be used to suggest the "*n*umber of things".

We can study algorithms and draw conclusions about how the implementation of the algorithm will behave. For example, there are many sorting algorithms that require roughly n^2 operations to arrange a list into its natural order. Other algorithms can accomplish the same task in $n * \log_2 n$ operations. There can be a large difference in the number of operations needed to complete these two different algorithms when n gets very large.

Some algorithms don't grow with n. For example, if a method performs a few additions and assignment operations, the time required to perform these operations does not change when n increases. These instructions are said to run in *constant time*. The number of operations can be described as a constant function f(n) = k, where k is a constant.

Most algorithms do not run in constant time. Often there will be a loop that executes more operations in relation to the size of the data variable such as searching for an element in a collection, for example. The more elements there are to locate, the longer it can take.

Computer scientists use different notations to characterize the runtime of an algorithm. The three major notations O(n), Ω(n), and Θ(n) are pronounced “big-O”, “big-Omega”, and “big-Theta”, respectively. The big-O measurement represents the upper bound on the runtime of an algorithm; the algorithm will never run slower than the specified time. Big-Omega is symmetric to big-O. It is a lower bound on the running time of an algorithm; the algorithm will never run faster than the specified time. Big-Theta is the tightest bound that can be established for the runtime of an algorithm. It occurs when the big-O and Omega running times are the same, therefore it is known that the algorithm will never run faster or slower then the time specified. This textbook will introduce and use only big-O.

When using notation like big-O, the concern is the *rate of growth* of the function instead of the precise number of operations. When the size of the problem is small, such as a collection with a small size, the differences between algorithms with different runtimes will not matter. The differences grow substantially when the size grows substantially.

Consider an algorithm that has a cost of $n^2 + 80n + 500$ statements and expressions. The upper bound on the running time is $O(n^2)$ because the larger growth rate function dominates the rest of the terms. The same is true for coefficients and constants. For very small values of n, the coefficient 80 and the constant 500 will have a greater impact on the running time. However, as the size grows, their impact decreases and the highest order takes over. The following table shows the growth rate of all three terms as the size, indicated by n, increases.

Function growth and weight of terms as a percentage of all terms as n increases

n	f(n)	n^2	80n	500
10	1,400	100 (7%)	800 (57%)	500 (36%)
100	18,500	10,000 (54%)	8,000 (43%)	500 (3%)
1000	1,0805,000	1,000,000 (93%)	80,000 (7%)	500 (0%)
10000	100,800,500	100,000,000 (99%)	800,000 (1%)	500 (0%)

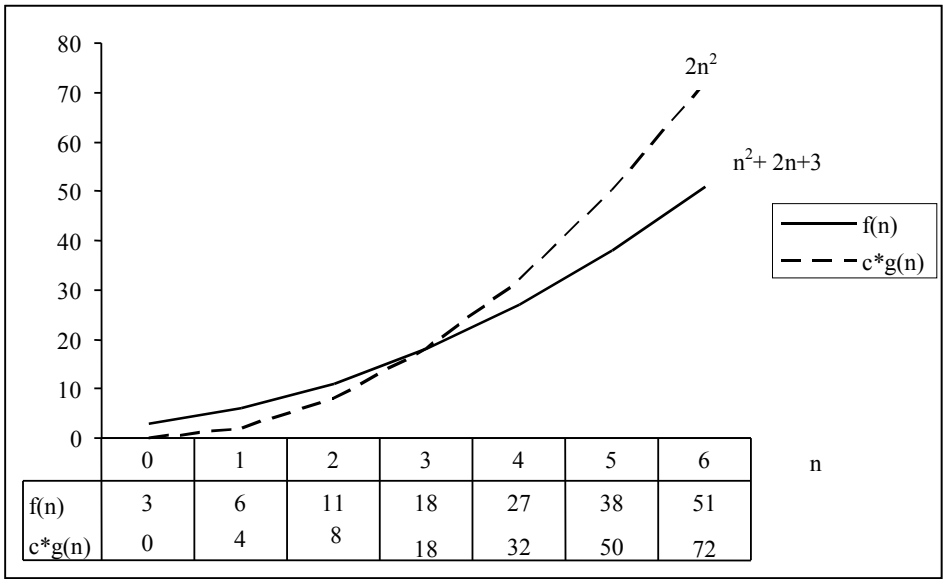
This example shows that the constant 500 has 0% impact (rounded) on the running time as n increases. The weight of this constant term shrinks to near 0%. The term 80n has some impact,

but certainly not as much as the term n^2, which raises n to the 2nd power. Asymptotic notation is a measure of runtime complexity when n is large. Big-O ignores constants, coefficients, and lower growth terms.

12.2 Big-O Definition

The big-O notation for algorithm analysis has been introduced with a few examples, but now let's define it a little further. We say that f(n) is O(g(n)) if and only if there exist two positive constants c and N such that f(n) <= c*g(n) for all n > N. We say that g(n) is an asymptotic upper bound for f(n). As an example, consider this graph where $f(n) = n^2 + 2n + 3$ and $g(n) = c*n^2$

Show that $f(n) = n^2 + 2n + 3$ is $O(n^2)$



	0	1	2	3	4	5	6
f(n)	3	6	11	18	27	38	51
c*g(n)	0	4	8	18	32	50	72

To fulfill the definition of big-O, we only find constants c and N at the point in the graph where c*g(n) is greater than f(n). In this example, this occurs when c is picked to be 2.0 and N is 4. The above graph shows that if n < N, the function g is at or below the graph of f. In this example, when n ranges from 0 through 2, g(n) < f(n). c*g(n) is equal to f(n) when c is 2 and n is 3 ($2*3^2$ ==18 as does $3^2+2*3+3$). And for all n >= 4, f(n) <= c*g(n). Since g(n) is larger than f(n) when c is 2.0 and N >= 4, it can be said that f(n) is O(g(n)). More specifically, f(n) is $O(n^2)$.

The g(n) part of these charts could be any of the following common big-O expressions that represent the upper bound for the runtime of algorithms:

Big-O expressions and commonly used names

O(1)	*constant (an increase in the size of the problem (n) has no effect)*
O(log n)	*logarithmic (operations increase once each time n doubles)*
O(n)	*linear*
O(n log n)	*n log n (no abbreviated name, also a computational biologist's AZ licence plate*
$O(n^2)$	*quadratic*
$O(n^3)$	*cubic*
$O(2^n)$	*exponential*

Properties of Big-O

When analyzing algorithms using big-O, there are a few properties that will help to determine the upper bound of the running time of algorithms.

Property 1, coefficients: If f(n) is x * g(n) then f(n) is O(g(n))
This allows the coefficient (x) to be dropped.
Example:
f(n) = 100*g(n)
then f(n) is O(n)

Property 2, sum: If $f_1(n)$ is O(g(n)) and $f_2(n)$ is O(g(n)) then $f_1(n) + f_2(n)$ is O(g(n))
This property is useful when an algorithm contains several loops of the same order.
Example:
$f_1(n)$ is O(n)
$f_2(n)$ is O(n)
then $f_1(n) + f_2(n)$ is O(n) + O(n), which is O(n)

Property 3, sum: If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$ then $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$. This property works because we are only concerned with the term of highest growth rate.
Example:
$f_1(n)$ is $O(n^2)$
$f_2(n)$ is O(n)
so $f_1(n) + f_2(n) = n^2 + n$, which is $O(n^2)$

Property 4, multiply: If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$ then $f_1(n) * f_2(n)$ is $O(g_1(n) * g_2(n))$. This property is useful for analyzing segments of an algorithm with nested loops.
Example:
$f_1(n)$ is $O(n^2)$
$f_2(n)$ is O(n)
then $f_1(n) * f_2(n)$ is $O(n^2) * O(n)$, which is $O(n^3)$

12.3 Counting Operations

We now consider one technique for analyzing the runtime of algorithms—approximating the number of operations that would execute with algorithms written in Java. This is the *cost* of the code. Let the cost be defined as the total number of operations that would execute in the worst case. The operations we will measure may be assignment statements, messages, and logical expression evaluations, all with a cost of 1. This is very general and does not account for the differences in the number of machine instructions that actually execute. The cost of each line of code is shown in comments. This analysis, although not very exact, is precise enough for this illustration. In the following code, the first three statements are assignments with a cost of 1 each.

Example 1

```
int n = 1000;                    // 1 instruction
int operations = 0;              // 1
int sum = 0;                     // 1
for (int j = 1; j <= n; j++) {   // 1 + (n+1) +  n
  operations++;                  // n
  sum += j;                      // n
}
```

The loop has a logical expression $j \leq n$ that evaluates $n + 1$ times. (The last time it is false.) The increment j++ executes n times. And both statements in the body of the loop execute n times. Therefore the total number of operations $f(n) = 1 + 1 + 1 + 1 + (n+1) + n + n + n = 4n + 5$. To have a runtime O(n), we must find a real constant c and an integer constant N such that $4n + 5 \leq cN$ for all $N > n$. There are an infinite set of values to choose from, for example $c = 6$ and $N = 3$, thus $17 \leq 18$. This is also true for all $N > 3$, such as when $N = 4$ ($21 \leq 24$) and when $N = 5$ ($25 < 30$). A simpler way to determine runtimes is to drop the lower order term (the constant 5) and the coefficient 4.

Example 2

A sequence of statements that does not grow with n is O(1) (constant time). For example, the following algorithm (implemented as Java code) that swaps two array elements has the same runtime for any sized array. $f(n) = 3$, which is O(1).

```
private void swap(String[] array, int left, int right)  {
  String temp = array[left];      // 1
  array[left] = array[right];     // 1
  array [right] = temp;           // 1
}
```

For a runtime O(1), we must find a real constant c and an integer constant N such that $f(n) = 3 \leq cN$. For example, when $c = 2$ and $N = 3$ we get $3 \leq 6$.

Example 3

The following code has a total cost of $6n + 3$, which after dropping the coefficient 6 and the constant 3, is O(n).

```
// Print @ for each n
for (int i = 0; i < 2 * n; i++)  // 1 + (2n+1) + 2n
  System.out.print("@");         // 2n+1
```

To have a runtime O(n), we must find a real constant c and an integer constant N such that $f(n) = 2n+1 \leq cN$. For example, $c = 4$ and $N = 3$ ($7 \leq 12$).

Example 4

Algorithms under analysis typically have one or more loops. Instead of considering the comparisons and increments in the loop added to the number of times each instruction inside the body of the loop executes, we can simply consider how often the loop repeats. A few assignments before or after the loop amount to a small constant that can be dropped. The following loop, which sums all array elements and finds the largest, has a total cost of $5n + 1$. The runtime once again, after dropping the coefficient 5 and the constant 1, is O(n).

```
double sum = 0.0;              // 1
double largest = a[0];         // 1
for (int i = 1; i < n; i++) {  // 1 + n + (n-1)
  sum += a[i];                 // n-1
  if (a[i] > largest)          // n-1
    largest = a[i];            // n-1, worst case: a[i] > largest always
}
```

Example 5

In this next example, two loops execute some operation n times. The total runtime could be described as O(n) + O(n). However, a property of big O is that the sum of the same orders of magnitude is in fact that order of magnitude (see big-O properties below). So the big-O runtime of this algorithm is O(n) even though there are two individual `for` loops that are O(n).

```
// f(n) = 3n + 5 which is O(n)
// Initialize n array elements to random integers from 0 to n-1
int n = 10;                                                 // 1
int[] a = new int[n];                                       // 1
java.util.Random generator = new java.util.Random();        // 1
for (int i = 0; i < n; i++)                                 // 2n + 2
  a[i] = generator.nextInt(n);                              // n


// f(n) = 5n + 3 which is O(n)
// Rearrange array so all odd integers in the lower indexes
int indexToPlaceNextOdd = 0;            // 1
for (int j = 0; j < n; j++) {           // 2n + 2
  if (a[j] % 2 == 1) {                  // n: worst case with all odds
    // Swap the current element into
    // the sub array of odd integers
    swap(a, j, indexToPlaceNextOdd);    // n
    indexToPlaceNextOdd++;              // n
  }
}
```

To reinforce that O(n) + O(n) is still O(n), all code above can be counted as f(n)= 8n + 8, which is O(n). To have a runtime O(n), use c = 12 and N = 4 where $10n+8 \leq cN$, or $40 \leq 48$.

Example 6

The runtime of nested loops can be expressed as the product of the loop iterations. For example, the following inner loop executes (n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1 times, which is n/2 operations. The outer loop executes the inner loop n-1 times. The inner loop executes (n-1)*(n/2) twice, which is $n^2 - n$ operations. Add the others to get $f(n) = 3n^2+4n-2$. After dropping the coefficient from n^2 and the lower order terms 4n and –2, the runtime is $O(n^2)$.

```
// Rearrange arrays so integers are sorted in ascending order
for (int top = 0; top < n - 1; top++) {        // 2n + 1
  int smallestIndex = top;                     //  n - 1
  for (int index = top; index < n; index++) {  // (n-1)*(2n)
    if (a[index] < a[smallestIndex])           // (n-1)*(n/2)
      smallestIndex = index;                   // (n-1)*(n/2) at worst
  }
  swap(a, top, smallestIndex);                 // 3n
}
```

To have a runtime $O(n^2)$, use c = 4 and N = 4 where $3n^2 + 4n - 2 \leq cN$, or $62 \leq 64$.

Example 7

If there are two or more loops, the longest running loop takes precedence. In the following example, the entire algorithm is $O(n^2) + O(n)$. The maximum of these two orders of magnitudes is $O(n^2)$.

```
int operations = 0;                          // 1
int n = 10000;                               // 1
// The following code runs O(n*n)
for (int j = 0; j < n; j++)                  // 2n+2
  for (int k = 0; k < n; k++)                // n*(2n+2)
    operations++;                            // n*(2n+2)


// The following code runs O(n)
for (int i = 0; i < n; i++)                  // 2n+2
  operations++;                              // n
```

Since $f(n) = 4n^2+9n+6 < cn^2$ for $c = 6.05$ when $N = 5$, $f(n)$ is $O(n^2)$.

Tightest Upper Bound

Since big-O notation expresses the notion that the algorithm will take no longer to execute than this measurement, it could be said, for example, that sequential search is $O(n^2)$ or even $O(2^n)$. However, the notation is only useful by stating the runtime as a tight upper bound. The tightest upper bound is the lowest order of magnitude that still satisfies the upper bound. Sequential search is more meaningfully characterized as $O(n)$.

Big-O also helps programmers understand how an algorithm behaves as n increases. With a linear algorithm expressed as $O(n)$, as n doubles, the number of operations doubles. As n triples, the number of operations triples. Sequential search through a list of 10,000 elements takes 10,000 operations in the worst case. Searching through twice as many elements requires twice as many operations. The runtime can be predicted to take approximately twice as long to finish on a computer.

Here are a few algorithms with their big-O runtimes.

- Sequential search (shown earlier) is $O(n)$
- Binary search (shown earlier) is $O(\log n)$
- Many sorting algorithms such as selection sort (shown earlier) are $O(n^2)$
- Some faster sort algorithms are $O(n \log n)$ — one of these (Quicksort) will be later
- Matrix multiplication is $O(n^3)$

Self-Check

12-1 Arrange these functions by order of growth from highest to lowest

$100*n^2$ 1000 2^n $10*n$ n^3 $2*n$

12-2 Which term dominates this function when n gets really big, n^2, 10n, or 100?

$n^2 + 10n + 100$

12-3. When n = 500 in the function above, what percentage of the function is the term?

12-4 Express the tightest upper bound of the following loops in big-O notation.

a)
```
int sum = 0;
int n = 100000;
```

b)
```
int sum = 0;
for (int j = 0; j < n; j++)
  for (int k = 0; k < n; k++)
     sum += j * k;
```

c)
```
for (int j = 0; j < n; j++)
  for (int k = 0; k < n; k++)
    for (int l = 0; l < n; l++)
      sum += j * k * l;
```

d)
```
for (int j = 0; j < n; j++)
  sum++;
for (int j = 0; j < n; j++)
  sum--;
```

e)
```
for (int j = 0; j < n; j++)
  sum += j;
```

f)
```
for (int j = 1; j < n; j *= 2)
  sum += j;
```

Search Algorithms with Different Big-Os

A significant amount of computer processing time is spent searching. An application might need to find a specific student in the registrar's database. Another application might need to find the occurrences of the string "data structures" on the Internet. When a collection contains many, many elements, some of the typical operations on data structures—such as searching—may become slow. Some algorithms result in programs that run more quickly while other algorithms noticeably slow down an application.

Sequential Search

This sequential search algorithm begins by comparing the first element in the array.

sequentially compare all elements, from index 0 to size-1 {
    if searchID equals ID of the object
        return a reference to that object
}
return null because searchID does not match any elements from index 0..size-1

If there is no match, the second element is compared, then the third, up until the last element. If the element being sought is found, the search terminates. Because the elements are searched one after another, in sequence, this algorithm is called **sequential** search. However since the worst case is a comparison of all elements and the algorithm is O(n), it is also known as **linear** search.

The binary search algorithm accomplishes the same task as sequential search. Binary search is more efficient. One of its preconditions is that the array must be sorted. Half of the elements can be eliminated from the search every time a comparison is made. This is summarized in the following algorithm:

***Algorithm:* Binary Search, use with sorted collections that can be indexed**

while the element is not found and it still may be in the array {
    Determine the position of the element in the middle of the array as middle
    If $array_{middle}$ equals the search string
        return $array_{middle}$
    If $array_{middle}$ is not the one being searched for:
        remove the half the sorted array that cannot contain the element form further searches
}

Each time the search element is compared to one array element, the binary search effectively eliminates half the remaining array elements from the search. This cuts the search field in half making binary search run O(log n)

When n is small, the binary search algorithm does not see a gain in terms of speed. However when n gets large, the difference in the time required to search for an element can make the difference between selling the software and having it unmarketable. Consider how many comparisons are necessary when n grows by powers of two. Each doubling of n would require potentially twice as many loop iterations for sequential search. However, the same doubling of n would only require potentially one more comparison for binary search.

Maximum number of comparisons for two different search algorithms

Power of 2	n	Sequential Search	Binary Search
2^2	4	4	2
2^4	16	16	4
2^8	128	128	8
2^{12}	4,096	4,096	12
2^{24}	16,777,216	16,777,216	24

As n gets very large, sequential search has to do a lot more work. The numbers above represent the maximum number of iterations necessary to search for an element. The difference between 24 comparisons and almost 17 million comparisons is quite dramatic, even on a fast computer. Let us analyze the binary search algorithm by asking, "How fast is Binary Search?"

The best case is when the element being searched for is in the middle—one iteration of the loop. The upper bound occurs when the element being searched for is not in the array. Each time through the loop, the "live" portion of the array is narrowed to half the previous size. The number of elements to consider each time through the loop begins with n elements (the size of the collection) and proceeds like this: n/2, n/4, n/8, ... 1. Each term in this series represents one comparison (one loop iteration). So the question is "How long does it take to get to 1?" This will be the number of times through the loop. Another way to look at this is to begin to count at 1 and double this count until the number *k* is greater than or equal to n.

$1, 2, 4, 8, 16, \ldots, k >= n$ or $2^0, 2^1, 2^2, 2^3, 2^4, \ldots, 2^c >= n$

The length of this series is c+1. The number of loop iterations can be stated as “2 to what power *c* is greater than or equal to n?” .

if n is 2, *c* is 1
if n is 4, *c* is 2
if n is 5, *c* is 3
if n is 100, *c* is 7
if n is 1024, *c* is 10
if n is 16,777,216, *c* is 24

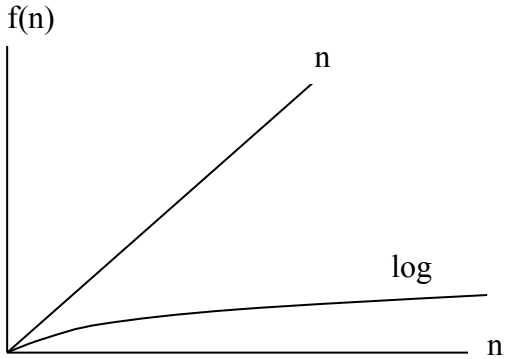
In general, as the number of elements to search (n) doubles, binary search requires only one more iteration to effectively remove half of the array elements from the search. The growth of this function is said to be **logarithmic**. Binary search is O(log n). The base of the logarithm (2) is not written, for two reasons:

- The difference between $\log_2 n$ and $\log_3 n$ is a constant factor and constants are not a concern.

- The convention is to use base 2 logarithms.

The following graph illustrates the difference between linear search, which is O(n), and binary search, which takes at most $\log_2 n$ comparisons.

Comparing O(n) to O(log n)



To further illustrate, consider the following experiment: using the same array of objects, search for every element in that array. Do this using both linear search and binary search. This experiment searches for every single list element. There is one O(n) loop that calls the binary search method with an O(log n) loop. Therefore, the time to search for every element using binary search indicates an algorithm that is O(n log n).

Searching for every element in the array (1.3 gig processor, 512 meg RAM):

Binary Search for every element O(n log n)				Sequential search for every element $O(n^2)$		
n	Binary Search #operations	average operations per search	total time in seconds	Sequential Search #operations	average operations per search	total time in seconds
100	588	5.9	0.00	5,050	50.5	0.0
1,000	9,102	9.1	0.01	500,500	500.5	0.3
10,000	124,750	12.4	0.30	5,0005,000	5,000.5	4.7
100,000	1,581,170	15.8	2.40	5,000,050,000	50,000.5	1,168.6

The time for sequential search also reflects a search for every element in the list. An O(n) loop calls a method that in turn has a loop that executes operations as follows (searching for the first element requires 1 comparison, searching for the second element requires 2 comparisons, and searching for the last two elements requires n-1 and n operations).

1 + 2 + 3 +, ... + n-2 + n-1 + n

This sequence adds up to the sum of the first n integers: n(n+1)/2. So when n is 100, 100(100+1)/2 = 5050 operations are required. The specific number of operations after removing the coefficient 1/2 is n*(n+1). Sequentially searching for every element in a list of size n is $O(n^2)$. Notice the large difference when n is 100,000: 1,168 seconds for the $O(n^2)$ algorithm compared to 4.5 seconds for the O(n log n) operation.

One advantage of sequential search is that it does not require the elements to be in sorted order. Binary search does have this precondition. This should be considered when deciding which searching algorithm to use in a program. If the program is rarely going to search for an item, then the overhead associated with sorting before calling binary search may be too costly. However, if

the program is mainly going to be used for searching, then the time expended sorting the list may be made up with repeated searches.

12.5 Example Logarithm Functions

Here are some other applications that help demonstrate how fast things grow if doubled and how quickly something shrinks if repeatedly halved.

1. Guess a Number between 1 and 100

Consider a daily number game that asks you to guess some number in the range of 1 through 1000. You could guess 1, then 2, then 3, all the way up to 1000. You are likely to guess this number in 500 tries, which grows in a linear fashion. Guessing this way for a number from 1 to 10,000 would likely require 10 times as many tries. However, consider what happens if you are told your guess is either too high, too low, or just right

Try the middle (500), you could be right. If it is too high, guess a number that is near the middle of 1..499 (250). If your initial guess was too low, check near middle of 501..1000 (750). You should find the answer in $2^c >= 1000$ tries. Here, c is 10. Using the base 2 logarithm, here is the maximum number of tries needed to guess a number in a growing range.

from 1..250, a maximum of $2^c >= 250$, c == 8
from 1..500, a maximum of $2^c >= 500$, c == 9
from 1..1000, a maximum of $2^c >= 1000$, c == 10

2. Layers of Paper to Reach the Moon

Consider how quickly things grow when doubled. Assuming that an extremely large piece of paper can be cut in half, layered, and cut in half again as often as you wish, how many times do you need to cut and layer until paper thickness reaches the moon? Here are some givens:

4. paper is 0.002 inches thick
5. distance to moon is 240,000 miles
6. 240,000 * 5,280 feet per mile * 12 inches per foot = 152,060,000,000 inches to the moon
7.

3. Storing Integers in Binary Notation

Consider the number of bits required to store a binary number. One bit represents two unique integer values: 0 and 1. Two bits can represent the four integer values 0 through 3 as 00, 01, 10, and 11. Three bits can store the eight unique integer values 0 through 7 as 000, 001, 010, ... 111. Each time one more bit is used twice as many unique values become possible.

4. Payoff for Inventing Chess

It is rumored that the inventor of chess asked the grateful emperor to be paid as follows: 1 grain of rice on the first square, 2 grains of rice on the next, and double the grains on each successive square. The emperor was impressed until later that month he realized that the 2^{64} grains of rice on the 64th square would be enough rice to cover the earth's surface.

Answers to Self-Check Questions

12-1 order of growth, highest to lowest

-1 2^n (2 to the nth power)	-3 $100*n^2$	-5 2*n
-2 n^3	-4 10*n	-6 1000

12-2 n^2 dominates the function

12-3 percentage of the function

n^2 = 98%
10n = 1.96%
100 = 0.0392%

12-4 tightest upper bounds

-a O(1)	-d O(n)
-b $O(n^2)$ On the order of n squared	-e O(n)
-c $O(n^3)$	-f O(log n)

Chapter 13

Collection Considerations

Goals

- Provide an overview of the major topics in a typical CS2 course
- Consider three data structures used in this textbook: arrays, the singly linked structure, and binary trees
- Observe that a Java interface can be used to specify a type to be implemented with different classes using different data structures
- Distinguish collection classes, data structures, and abstract data types
- Consider two different ways to have the same collection class store a collection of any type: 1) Object parameters and 2) Generics

First some definitions that will be repeated later:

Abstract Data Type (ADT)

- A set of data values and associated operations that are precisely specified independent of any particular implementation.
    - Examples: Bag, Set, List, Stack, Queue, Map

Collection Class

- A Java language construct for encapsulating the data and operations.
    - Examples: ArrayBag, ArrayList, LinkedList, TreeSet, HashMap

Data Structure

- An organization of information in the computer's memory.
    - Examples: arrays, the singly linked structure, binary trees, hash tables

---

13.1 ADTs, Collection Classes, Data Structures

An **abstract data type** (ADT) describes a set of data values and associated operations that are precisely specified independent of any particular implementation. An abstract data type can be specified using axiomatic semantics. As one example, here is the Bag ADT as described by the National Institute of Standards and Technology (NIST)[1].

[1] `http://www.nist.gov/dads/HTML/bag.html`

Bag

Definition: An unordered collection of values that may have duplicates.

Formal Definition: A bag has a single query function, occurencesOf(v, B), which tells how many copies of an element are in the bag, and two modifier functions, add(v, B) and remove(v, B). These may be defined with axiomatic semantics as follows.

1. new() returns a bag
2. occurencesOf(v, new()) = 0
3. occurencesOf(v, add(v, B)) = 1 + occurencesOf(v, B) [2]
4. occurencesOf(v, add(u, B)) = occurencesOf(v, B) if v ≠ u
5. remove(v, new()) = new()
6. remove(v, add(v, B)) = B
7. remove(v, add(u, B)) = add(u, remove(v, B)) if v ≠ u

where B is a bag and u and v are elements.

The predicate isEmpty(B) may be defined with the following additional axioms:

8. isEmpty(new()) = true
9. isEmpty(add(v, B)) = false

Also known as multi-set.
Note: A bag, or multi-set, is a set where values may be repeated. Inserting 2, 1, 2 into an empty set gives the set {1, 2}. Inserting those values into an empty bag gives {1, 2, 2}.

There are other ways to specify ADTs including the Java interface.

Java Interfaces

A Java `interface` begins with a heading that is similar to a Java **`class`** heading except the keyword **`interface`** is used. The interface consists of method headings; just the headings, no implementation of methods. Interface can not have constructors or instance variables either.

The Bag type described above with axiomatic expressions can also be specified with the following Java `interface` (Note: The parameter `Object`, which is covered in the next section represents the type of element to be stored in the Bag; the type of v in NIST's bag above):

```
/**
 * This Java interface specifies the operations of a Bag ADT.
 */
public interface Bag {

  // Return true if there are no elements in this Bag.
  public boolean isEmpty();

  // Add a v to this collection.
  public void add(Object v);
```

[2] This definition shows a bag B is passed as an argument. In an object-oriented language you send a message to an object of type B as in `aBag.add("New Element);` rather than `add("New Element", aBag);`

```
  // Return how often the value v exists in this StringBag.
  public int occurencesOf(Object v);

  // If an element that equals v exists, remove one occurrence of it from this
  // Bag and return true. If occurencesOf(v) == 0, simply return false.
  public boolean remove(Object v);
}
```

One or more Java classes will implement a Java `interface`. When a class implements an ADT whose main purpose is to store a collection of elements—such as Bag—it is called a collection class.

Collection Classes

A **collection class** is a class whose main purpose is to store a collection of elements. Most collection classes are first described as a Java interface. However, interfaces cannot be instantiated. There must be one or more Java classes that *implement* the interface. This is accomplished with the keyword **`implements`** followed by the name of the interface `Bag`.

```
/**
 * A collection class for storing a Bag of string elements.
 * This class must have all of the method specified in interface Bag.
 *
 * Note: This version has only method stubs that do not work but do compile.
 */
public class ArrayBag implements Bag {

  // Construct an empty StringBag object
  public ArrayBag() {
    // TODO Implement this method stub.
  }

  // Return true if there are no elements in this Bag
  public boolean isEmpty() {
    // TODO Implement this method stub.
    return false;
  }

  // Add an element with the value of v to this collection.
  public void add(Object v) {
    // TODO Implement this method stub.
  }

  // Return how often the value v exists in this StringBag.
  public int occurencesOf(Object v) {
    // TODO Implement this method stub.
    return -1;
  }

  // If an element that equals v exists, remove one occurrence of v from this
  // Bag and return true. If occurencesOf(v) == 0, simply return false.
  public boolean remove(Object v) {
    // TODO Implement this method stub.
    return false;
  }
}
```

By making a class implement an interface, all methods of that interface must be part of the class. If any method is missing, or a signature varies slightly, the class will not compile. This is the first step in implementing an interface—getting the class to compile.

To make the entire class compile, it is therefore necessary to have non-void methods such as `public boolean isEmpty()` return a value of the correct type. In the case of `isEmpty`, the return value must be `true` or `false`. The `boolean` value `false` was arbitrarily chosen just to get the method to compile. For an `int` return type, it could be any of the 4.2 billion or so valid integers. In the case of `occurencesOf`, -1 was arbitrarily chosen to get the method to compile

The methods above are written as method stubs—methods that do not work but do compile. The `Bag` interface and the beginning of the collection class `ArrayBag` that implements that `Bag` interface now compile. This means we could write tests, even if the assertions in those `@Test` methods will not pass. This allows us to use assertions to further describe the behavior of how the operations should work (add two occurrences of "Sam" and assert the `ArrayBag` has 2).

```
/**
 * This unit test shows a further specification of the Bag ADT. It can also be
 * used later to develop ArrayBag and test that the new class works correctly.
 */
import static org.junit.Assert.*;
import org.junit.Test;

public class ArrayBagTest {

  @Test
  public void testIsEmptyWithOneAdd() {
    ArrayBag names = new ArrayBag();
    assertTrue(names.isEmpty());
    names.add("Kim");
    assertFalse(names.isEmpty());
  }

  @Test
  public void testOccurencesOfWithOneElement() {
    ArrayBag names = new ArrayBag();
    assertEquals(0, names.occurencesOf("Kim"));
    names.add("Kim");
    assertEquals(1, names.occurencesOf("Kim"));
  }

  @Test
  public void testOccurencesOfWhenMoreThanOneExists() {
    ArrayBag names = new ArrayBag();
    names.add("Sam");
    names.add("Devon");
    names.add("Sam");
    assertEquals(0, names.occurencesOf("Not here"));
    assertEquals(1, names.occurencesOf("Devon"));
    assertEquals(2, names.occurencesOf("Sam"));
  }

  @Test
  public void testRemove() {
    ArrayBag names = new ArrayBag();
    names.add("Sam");
    names.add("Chris");
    names.add("Devon");

    // Return false if the element does not occur
    assertFalse(names.remove("Not here"));
```

```
    // Remove "Sam" successfully, then try again.
    assertTrue(names.remove("Sam"));
    assertEquals(0, names.occurencesOf("Sam"));
    assertFalse(names.remove("Sam"));
    assertEquals(0, names.occurencesOf("Sam"));

    // Remove all other elements
    assertEquals(1, names.occurencesOf("Chris"));
    assertTrue(names.remove("Chris"));
    assertEquals(0, names.occurencesOf("Chris"));

    // Only one element left
    assertFalse(names.isEmpty());
    assertEquals(1, names.occurencesOf("Devon"));
    assertTrue(names.remove("Devon"));
    assertEquals(0, names.occurencesOf("Devon"));
    assertTrue(names.isEmpty());
  }
}
```

In addition to writing the class and method stubs and a unit test with assertions to provide example usage and how the operations are expected to work, we still need to determine what data structure to use (Hint: Because the class is named `ArrayBag`, this chapter will use the array data structure as the example instance variable for storing the elements of the collection, consider the others a preview of things to come whose presence indicates we have more than arrays to store collections).

Data Structures

A **data structure** is a way of storing data on a computer so it can be used efficiently. There are several data structures available for storing collections of data. Some are more appropriate than others, depending on how you need to manage your data. Although not a complete list, here are some of the data structures used to store a collection of elements—they will become instance variables inside collection classes.

- arrays (1D, 2D, 3 dimensions)
- the singly linked structure
- the binary tree
- hash tables

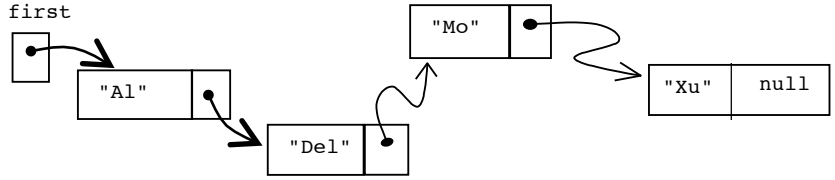
Data can be stored in contiguous memory with arrays or in non-contiguous memory with linked structures. Arrays allow you to reserve memory where each element can be physically located next to its predecessor and successor. Any element can be directly changed and accessed through an index. It is this random access to elements in the array that makes arrays attractive.

```
String[] data = new String[5];
data[0] = "Al";
data[1] = "Di";
data[2] = "Mo";
data[3] = "Xu";
```

`data` (where `data.length` == 5):

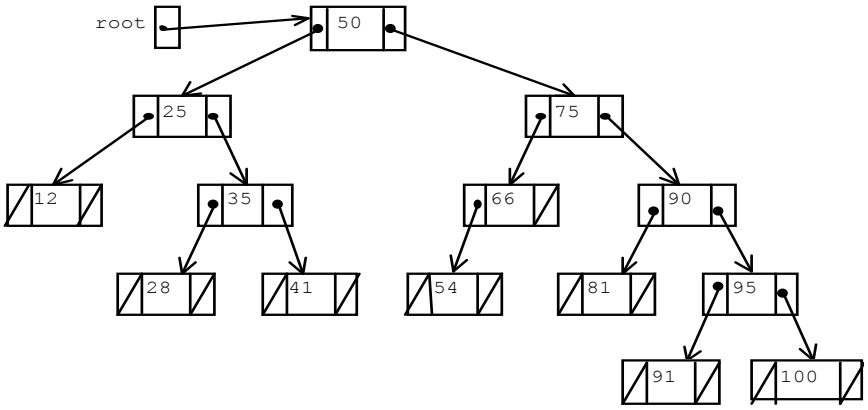
[0]	[1]	[2]	[3]	[4]
"Al"	"Di"	"Mo"	"Xu"	null

A **singly linked structure** contains nodes that store a reference to an element and a reference to another node. The reference to another node may be null to indicate there are no more elements stored.



You will see that both of these storage mechanisms — arrays and linked structures — implement the same List ADT in subsequent chapters.

You will also see how binary trees store data that can be added, removed, and found more quickly. The following picture indicates a linked structure where each node contains a reference to data (integers are used here to save space) and references to two other nodes, or to nothing (shown as diagonal lines).



The hash table is another data structure uses a key such as a student ID, employee number, or Social Security Number. The values can be found quickly using a hash function that converts the string key into an array index. This hash table can store up to 754 `BankAccount` objects (`null` as a key means a new element could be added later in that array index).

Array Index	Key	Value mapped to the key (null or 4 objects shown)
[0]	"1023"	"Devon" 512.99
[1]	null	null
[2]	"5462"	"Ali" 0.00
[3]	null	null
[4]	"3343"	"Taylor" 7962.34
…		
[753]	"0930"	"Chris" 567.89

Although hash tables are beyond the scope of this book, you will see collection classes using the other three data structures to implement abstract data types with a java class.

13.2 Generic Collections with `Object[]`

If you wanted to store a collection of `Integer` objects you could have an `IntegerBag` class. To store a collection of Strings, a new `StringBag` class would have to be written, or a new class for any type such as `BankAccountBag`, `DoubleBag`, `StudentBag`, and so on. Rather than implementing a new collection class for all types, Java provides two approaches to allow for just one collection class to store any type of elements:

1. Store references to `Object` objects rather than one particular type (this section 13.2)
2. Use Java generics with a type parameter like `<E>` (section 13.3)

We'll consider the first option now, which for better or worse, requires knowledge of Java's `Object` class, inheritance, and casting.

Java's `Object` class has one constructor (no arguments) and 11 methods, including `equals` and `toString`. All classes in Java extend the `Object` class or another class that extends `Object`. There is no exception to this. All classes inherit the methods of the `Object` class (unless overridden).

One class inherits the methods and instance variables of another class with the keyword `extends`. For example, the following class heading explicitly states that the `EmptyClass` class inherits all of the method of class `Object`. If a class heading does not have `extends`, the compiler automatically adds `extends Object`.

```
// This class extends Object even if extends Object is omitted
public class EmptyClass extends Object {

}
```

Even though this `EmptyClass` defines no methods, a programmer can construct and send messages to `EmptyClass` objects. This is possible because a class that extends `Object` inherits (obtains) `Object`'s methods. Here are two of the methods that `EmptyClass` inherits from `class Object`:

Two Methods of the `Object` Class

- `toString` returns a `String` that is the class name concatenated with the at symbol `@` and a the hexadecimal (base 16) representation of the object HashCode value used to find it quickly while the program is running.
- `equals` returns `true` if both the receiver and argument reference the same object.

Additionally, a class that does not declare a constructor is given a default constructor. A default constructor is one that takes no arguments (no parameters). This is automatically supplied in order to ensure that `Object`'s constructor will be called to perform operations such as allocating memory for objects at runtime. The following class definition is equivalent to that shown above.

```
// This class extends Object implicitly
public class EmptyClass {

  public EmptyClass() {
    super();  // Explicitly call the constructor of the superclass: Object
  }
}
```

This following @Test method shows these two used by a class that extends class `Object`.

```
@Test
public void showInheritance () {
  EmptyClass one = new EmptyClass();
  EmptyClass two = new EmptyClass();

  System.out.println(two.toString());
  System.out.println(one.toString());

  assertFalse(one.equals(two)); // one and two refer to two unique objects
  one = two;                    // Assign one reference to the other
  assertTrue(one.equals(two)); // one and two now refer to the same object

  System.out.println("\nafter assignment--->\n");
  System.out.println(two.toString());
  System.out.println(one.toString());
}
```

Output

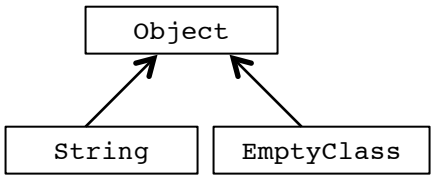
```
EmptyClass@60765a16
EmptyClass@7a148bd3

after assignment--->

EmptyClass@60765a16
EmptyClass@60765a16
```

Notice that at first, `one` and `two` are not `equals`—`one` and `two` refer to two different `EmptyClass` objects. After the assignment, both `one` and `two` refer to the same object. The reference values are now `equals` as indicated by the `toString` values.

One-way Assignment Compatibility

Because all classes extend Java's `Object` class, a reference to any type of object can be assigned to an `Object` reference variable. For example, consider the following valid code that assigns a `String` object and an `EmptyClass` object to two different reference variables of type `Object`:

```
String aString = new String("first");
// Assign a String reference to an Object reference
Object obj1 = aString;

EmptyClass one = new EmptyClass();
// Assign an EmptyClass reference to an Object reference
Object obj2 = one;
```



Java's one-way assignment compatibility means that you can assign a reference variable to the class that it extends. However, you cannot directly assign in the other direction. For example, an `Object` reference cannot be directly assigned to a `String` reference.

```
Object obj = new Object();
String str = obj;
```

Type mismatch: cannot convert from Object to String

This compile time error occurs because the compiler recognizes that `obj` is a reference to an `Object` that cannot be assigned down to a `String` reference variable.

Casting

Once a reference is stored into an `Object` reference variable, there is usually a need to get that reference back at some point. In this attempt, the compiler reports an error.

```
Object ref = "A string";
// Later we want to send messages to the string
String aString = ref;
```

Type mismatch: cannot convert from Object to String

Even though `ref` stores a reference to a `String`, at compile time, the compilers considers `ref` to be of type `Object` and `aString` of type `String`. The compiler does not allow this assignment of `Object` to `String`. The compiler has been trained to report an error when an attempt is made to assign down the inheritance hierarchy. Since we know that `ref` does indeed store a reference to a `String`, we have to promise the compiler that this is true. Some would say we *trick* the compiler with a **cast**.

To cast from one type to another, enclose the class name with what you know the class to be in parentheses `(String)` and place it before the reference to the `Object` reference `ref`.

```
Object ref = "A string";
// Later we want to send messages to the string
String aString = (String)ref;
// Many call this a cast but it really is a way to trick the compiler.
```

As long as the type we are casting `Object` to is indeed the correct type at runtime, everything is okay. However, if we trick ourselves in addition to tricking the compiler, the program can terminate with a runtime error.

```
Object ref = 1234;
// Later we want to send messages to the string
String aString = (String)ref;
```

java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String

In this case, the code is attempting to case an integer to a String, which is a ClassCastException.

Array of Object objects

It is also possible to have an array of references to this `Object` type. This allows any type element to be stored into any array location when the array is an `Object[]`:

```
Object[] data = new Object[10];
data[0] = "A string";
data[1] = 123;
data[2] = 4.56;
data[3] = 'C';
for(int i = 0; i < 4; i++) {
  System.out.println(data[i] + " ");
}
```

Output

```
A string
123
4.56
C
```

What is awkward about this is the fact that we can't always be sure what type is where. For example, this code compiles, but generates a runtime error and shuts the program down.

```
  assertEquals("A STRING", ((String)data[1]).toUpperCase());
```

java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String

We promised the compiler `data[1]` stores a reference to a String at runtime, but `data[1]` actually stores an int. In the next section, we will see how a collection can be made generic and avoid this situation and the ugly cast code.

Self-Check

13-1 Place a check mark √ in the comment after assignment statement that compiles (or leave blank).

```
Object anObject = new Object();
String aString = "abc";
Integer anInteger = new Integer(5);
anObject = aString;    // ______
anInteger = aString;   // ______
anObject = anInteger;  // ______
anInteger = anObject;  // ______
```

13-2 Place a check mark √ in the comment after assignment statement that compiles (or leave blank).

```
Object anObject = new String("abc");
Object anotherObject = new Integer(50);
Integer n = (Integer) anObject;          // ______
String s = (String) anObject;            // ______
anObject = anotherObject;                // ______
String another = (String) anotherObject; // ______
Integer anotherInt = (Integer) anObject; // ______
```

13-3 Which statements generate compile time errors?

```
Object[] elements = new Object[4]; // a.
elements[0] = 12;                  // b.
elements[1] = "Second";            // c.
elements[2] = 4.5;                 // d.
elements[3] = new Point(5, 6);     // e.
```

An Entire Collection Class

The `Bag` interface and the `ArrayBag` class used `Object` parameters in the `add` and `remove` methods. This is so we can use an array of `Object` type objects (`Object[]`) as the instance variable to store any type element. Because we can assign a reference of any type to an `Object` parameter, we can use this one class to store a collection of any type of value. One class for any type, rather than a separate collection class to store collections of every type.

```
/**
 * A collection class for storing a Bag of any type of elements.
 */
public class ArrayBag implements Bag {

  // This implementation of Bag uses an array data structure
  private Object[] data;
  int n = 0;

  // Construct an empty StringBag object with the capacity to store 20 elements
  public ArrayBag() {
    data = new Object[20];
    n = 0;
  }
```

```
  // Return true if there are no elements in this Bag
  public boolean isEmpty() {
    return n == 0;
  }

  // Add an element to this collection
  public void add(Object element) {
    data[n] = element;
    n++;
  }

  // Return how often the value element exists in this StringBag
  public int occurencesOf(Object element) {
    int result = 0;
    for (int i = 0; i < n; i++) {
      if (element.equals(data[i]))
        result++; // Found one that equals element
    }
    return result;
  }

  // If an value in data equals elements, remove one occurrence of it from this
  // Bag and return true. If occurencesOf(element) == 0, simply return false.
  public boolean remove(Object element) {
    for (int i = 0; i < n; i++) {
      if (element.equals(data[i])) {
        // Replace element to be removed with the one at the end of the array
        data[i] = data[n - 1];
        // And reduce the number of meaningful elements by 1
        n--;
        return true;
      }
    }
    // Must be that the element did not "equals" anything in this bag
    return false;
  }
}
```

This design allows one class to store collections with any type of elements:

```
  @Test
  public void testGenericity() {
    // A Bag of strings
    ArrayBag names = new ArrayBag();
    names.add("Taylor");
    names.add("Kim");
    assertEquals(1, names.occurencesOf("Kim"));

    // A Bag of integers
    ArrayBag testScores = new ArrayBag();
    testScores.add(99);
    testScores.add(77);
    testScores.add(88);
    testScores.add(88);
    assertEquals(2, testScores.occurencesOf(88));
  }
```

A class that implements an interface must have the methods of the interface. That class can also add extra methods. Consider adding a `get` method to class `ArrayBag` so we can look at all elements in the bag. To allow one collection class to store any type of element, we cannot return String or int. A method that returns a reference to the element in the collection would have to return an `Object` reference.

```
public Object get(int index) {
  return data[index];
}
```

This approach requires a cast with every `get` message. You have to know the type stored.

```
@Test
public void testGet() {
  ArrayBag strings = new ArrayBag();
  strings.add("abc");
  strings.add("def");
  assertEquals("ABC", ((String) strings.get(0)).toUpperCase());
  assertEquals("DEF", ((String) strings.get(1)).toUpperCase());
}
```

With this approach, programmers always have to cast. Java software developers had complained about this for many years (before Java 5). With this approach, you also have to be wary of runtime exceptions. For example, even though the following code compiles, when the test runs, a runtime error occurs.

```
@Test
public void testGet() {
  ArrayBag strings = new ArrayBag();
  strings.add("abc");
  strings.add(12345);  // <-- Notice that Java lets us add an integer
  assertEquals("ABC", ((String) strings.get(0)).toUpperCase());
  assertEquals("DEF", ((String) strings.get(1)).toUpperCase());
}
```

java.lang.ClassCastException: java.util.Integer

`strings.get(1)` returns a reference to an integer, which the runtime treats as a `String` in the cast. A `ClassCastException` occurs because a String cannot be cast to an integer. In a later section, Java Generics will be shown as a way to have a collection store a specific type. One collection class is all that is needed, and the casting and runtime errors will disappear.

Collections of Primitive Types

Collections of the primitive types such `int`, `double`, `char` can also be stored in a generic class. The type parameter could be one of Java's "wrapper" classes (or had to be wrapped before Java 5). Java has a "wrapper" class for each primitive type named `Integer`, `Double`, `Character`, `Boolean`, `Long`, `Short`, and `Float`. A wrapper class does little more than allow a primitive value to be viewed as a reference type that can be assigned to an Object reference. A `GenericList` of integer values can be stored like this:

```
GenericList tests = new GenericList();
tests.add(new Integer(79));
tests.add(new Integer(88));
```

However, integer values can also be added like this:

```
tests.add(76);
tests.add(100);
```

Java now allows primitive integers to be treated like objects through the process known as autoboxing.

Autoboxing / Unboxing

Before Java 5, to treat primitive types as reference types, programmers were required to "box" primitive values in their respective "wrapper" class. For example, the following code was used to assign an `int` to an `Object` reference.

```
Integer anInt = new Integer(123); // Wrapper class needed for an int
tests.add(anInt);                 // to be stored as an Object reference
```

To convert from reference type back to a primitive type, programmers were required to "unbox" by asking the `Integer` object for its `intValue` like this:

```
int primitiveInt = anInt.intValue();
```

Java 5.0 automatically performs this boxing and unboxing.

```
Integer anotherInt = 123;              // autobox 123 as new Integer(123)
int anotherPrimitiveInt = anotherInt;  // unboxed automatically
```

This allows primitive literals to be added. The autoboxing occurs automatically when assigning the int arguments 79 and 88 to the `Object` parameter of the `add` method.

```
GenericList tests = new GenericList();
tests.add(79);
tests.add(88);
test.add(new Integer(99));  // No longer need to construct explicitly
```

However, with the current implementation of `GenericList`,

```
public Object get(int atIndex)  {
  return elements[index];
}
```

we still have to cast the return value from this `get` method. The compiler sees the return type `Object` that must be cast to whatever type of value happens to stored at `elements[index]`:

```
Integer anInt = (Integer)tests.get(0);
```

Self-Check

13-4 Place a check mark √ in the comment after assignment statement that compiles (or leave blank).

```
Integer num1 = 5;                // _______
Integer num2 = 5.0;              // _______
Object num3 = 5.0;               // _______
int num4 = new Integer(6);       // _______
double num5 = new Double(7.7);   // _______
Integer num6 = new Double(8.8);  // _______
Integer num7 = 9;                // _______
Double num8 = 9.99;              // _______
```

13.3 Java Generics with Type Parameters

The manual boxing, the cast code, and the problems associated with collections that can accidentally add the wrong type element are problems that all disappeared with the release of Java 5 in 2004. Now the programmer can specify the one type that should be stored by passing the type as an argument to the collection class. Type parameters and arguments parameters are enclosed between < and >. Now these safer interface and collection classes look like this where E is the type parameter:

```
// Shortened interface to highlight the new type parameter E
public interface Bag<E> {
  public boolean isEmpty();
  public void add(E element);
  public int occurencesOf(E element);
  public boolean remove(E element);
}
```

```
// Shortened class to highlight the new type parameter E
public class ArrayBag<E> {

  private Object[] elements;
  private int n;

  public ArrayBag() {
    elements = new Object[10];
    n = 0;
  }

  public void add(E element) {
    elements[n] = elementToAdd;
    n++;
  }

  // . . .

  public E get(int index) {
    return (E)elements[index];
  }
}
```

```
import static org.junit.Assert.*;
import org.junit.Test;

public class ArrayBagTest {
  // Test highlighting passing type arguments String and Integer
  @Test
  public void testGenericity() {
    ArrayBag<String> names = new ArrayBag<String>();
    names.add("Taylor");
    names.add("Kim");
    ArrayBag<Integer> testScores = new ArrayBag<Integer>();
    testScores.add(99);
    testScores.add(77);
    testScores.add(88);
  }
}
```

In the `add` method, `Object` is replaced with `E` so only elements of the type argument—`String`, or `Integer` above—can be added to that collection.

The compiler catches accidental `add` messages using the wrong type of argument. For example, the following three attempts to add the wrong type generate ***compiletime*** errors (not runtime errors later. This is a good thing. It is better than waiting until runtime when the program would otherwise terminate.

```
names.add(999);
```

The method add(String) in the type ArrayBag<String> is not applicable for the arguments (int)

```
testScores.add("a String");
```

The method add(Integer) in the type ArrayBag <Integer> is not applicable for the arguments (String)

You also don't have to manually box primitives during `add` messages.

```
testScores.add(new Integer(89));
```

Nor do you have to cast the return values

```
names.add("First");
names.add("Second");
String noCastToStringNeeded = strings.get(0);
noCastToStringNeeded = strings.get(1);
```

because the cast is now done in the `get` method.

```
public E get(int index) {
  return (E)elements[index];
}
```

And in the case of `Integer`, the `Integer` object is automatically unboxed to a primitive `int`.

```
int sum = 0;
for (int i = 0; i < 3; i++) {
  sum += tests.get(i); // no cast needed because get already casts to (E)
}
```

Using Java generic type arguments does require extra syntax when constructing a collection (two sets of angle brackets and the type to be stored twice). However the benefits include much less casting syntax, the ability to have collections of primitives where the primitives appear to be objects. We gain the type safety that comes from allowing the one type of element to be maintained by the collection. The remaining examples in this textbook will use Java Generics with < and > rather than having parameters and return types of type Object.

Self-Check

13-5 Give the following code, print all elements in uppercase. Hint no cast required before sending a message.

```
ArrayBag<String> strings = new ArrayBag <String>();
strings.add("lower");
strings.add("Some UpPeR");
strings.add("ALL UPPER");
```

In Summary

- The Java interface specifies a data type abstractly; interface doe not implement methods.
- The collection class implementing that interface can be instantiated.
- The implementing class must implement all methods of the interface and can add more methods if so desired.
- The data structure used in the collection class can vary—an array, a linked structure, a tree, or a hash table. Each has advantages and disadvantages.
- The same interface is sometimes implemented by two or more classes to allow the same collection to have the same exact methods, but with a data structure that best suits an application. Sometimes an array is better, sometimes a linked structure is better.

In the next chapter, `interface OurList` describe the important list ADT. The interface will be implemented using the array data structure. In the chapter that follows, the same exact interface will be implemented using a completely different data structure: the singly linked list data structure.

Answers to Self-Check Questions

13-3 None

13-1
```
anObject = aString;     // __x___
anInteger = aString;    // Can't assign String to Integer
anObject = anInteger;   // __x___
anInteger = anObject;   // Can;t assign Obect to Integer
```

13-2
```
Integer n = (Integer) anObject;          // __x___
String s = (String) anObject;            // __x___
anObject = anotherObject;                // __x___
String another = (String) anotherObject; // __x___
Integer anotherInt = (Integer) anObject; // __x___
```

13-3 None

13-4
```
Integer num1 = 5;                 // __X___
Integer num2 = 5.0;               // ______
Object num3 = 5.0;                // __X___
int num4 = new Integer(6);        // __X___
double num5 = new Double(7.7);    // __X___
Integer num6 = new Double(8.8);   // ______
Integer num7 = 9;                 // __X___
Double num8 = 9.99;               // __X___
```

13-5
```
for (int i = 0; i < 3; i++) {
  System.out.println(strings.get(i).toUpperCase());
}
```

Chapter 14

A List ADT

This chapter defines an interface that represents a List abstract data type. The class that implements this interface uses an array data structure to store the elements. In the next chapter, we will see this very same interface implemented using a completely different data structure—the singly linked structure.

Goals

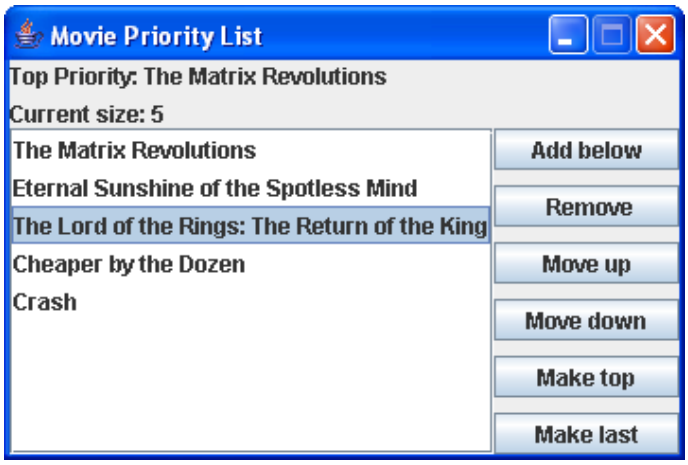
- Introduce a List ADT as a Java interface
- Implement an interface
- Shift array elements during insertions and removals
- Have methods throw exceptions and write tests to ensure that the methods do throw them.

14.1 A List ADT

A **list** is a collection where each element has a specific position—each element has a distinct predecessor (except the first) and a distinct successor (except the last). A list allows access to elements through an index. The list interface presented here supports operations such as the following:

- add, get, or remove an element at specific location in the list
- find or remove an element with a particular characteristic

From an *application* point of view, a list may store a collection of elements where the index has some importance. For example, the following interface shows one view of a list that stores a collection of DVDs to order. The DVD at index 0, "The Matrix Revolutions", has the top priority. The DVD at index 4 has a lower priority than the DVD at index 3. By moving any "to do" item up or down in the list, users reprioritize what movies to get next. Users are able to add and remove DVDs or rearrange priorities.



From an *implementation* point of view, your applications could simply use an existing Java collection class such as `ArrayList<E>` or `LinkedList<E>`. As is customary in a second level course in computer science, we will be implementing our own, simpler version, which will

- enhance your ability to use arrays and linked structures (required in further study of computing).
- provide an opportunity to further develop programming skills: coding, testing, and debugging.
- help you understand how existing collection classes work, so you can better choose which one to use in programs.

Specifying ADTs as Java Interfaces

To show the inner workings of a collection class (first with an array data structure, and then later with a linked structure), we will have the same interface implemented by two different classes (see Chapter 15: A Linked Structure). This interface, shown below, represents one abstract view of a list that was designed to support the goals mentioned above.

The interface specifies that implementing classes must be able to store any type of element through Java generics—`List<E>`, rather than `List`. One alternative to this design decision is to write a List class each time you need a new list of a different type (which could be multiple classes that are almost the same). You could implement a class for each type of the following objects:

```
// Why implement a new class for each type?
StringList stringList = new StringList();
BankAccountList bankAccountList = new BankAccountList();
DateList dateList = new DateList();
```

An alternative was shown with the `GenericList` class shown in the previous chapter. The method heading that adds an element would use an `Object` parameter and the `get` method to return an element would have an `Object` return type.

```
// Add any reference type of element (no primitive types)
public void add(Object element);

// Get any reference type of element (no primitive types)
public Object get(int index);
```

Collections of this type require the extra effort of casting when getting an element. If you wanted a collection of primitives, you would have to wrap them. Additionally, these types of collections allow you to add any mix of types. The output below also shows that runtime errors can occur because any reference type can be added as an element. The compiler approves, but we get a runtime exception.

```
GenericList list = new GenericList();
list.add("Jody");
list.add(new BankAccount("Kim", 100));
for (int i = 0; i < list.size(); i++) {
  String element = (String) list.get(i); // cast required
  System.out.println(element.toUpperCase());
}
```

Output:

```
JODY
Exception in thread "main" java.lang.ClassCastException: BankAccount
```

The preferred option is to focus on classes that have a type parameter in the heading like this

```
public class OurList<E>  // E is a type parameter
```

Now `E` represents the type of elements to that can be stored in the collection. Generic classes provide the same services as the raw type equivalent with the following advantages:

- require less casting
- can store collections of any type, including primitives (at least give the appearance of)
- generate errors at compile time when they are much easier to deal with
- this approach is used in the new version of Java's collection framework

Generic collections need a type argument at construction to let the compiler know which type E represents. When an `OurList` object is constructed with a `<String>` type argument, every occurrence of `E` in the class will be seen as `String`.

```
// Add a type parameter such as <E> and implement only one class
OurList<String> sl = new OurArrayList<String>();
OurList<BankAccount> bl = new OurArrayList<BankAccount>();
OurList<Integer> dl = new OurArrayList<Integer>();
```

Now an attempt to add a `BankAccount` to a list constructed to only store strings

```
sl.add(0, new BankAccount("Jody", 100));
```

results in this compiletime error:

The method add(int, String) in the type OurList<String> is not applicable for the arguments (int, BankAccount)

14.2 A List ADT Specified as a Java interface

Interface `OurList` specifies a reduced version of Java's `List` interface (7 methods instead of 25). By design, these methods match the methods of the same name found in the two Java classes that implement Java's `List` interface: `ArrayList<E>` and `LinkedList<E>`.

```
/**
  * This interface specifies the methods for a generic List ADT.
  * It is designed to be with a type parameter so any type element
  * can be stored.  Some methods will be implemented with an array
  * data structure in this chapter and then as a linked structure
  * in the chapter that follows.
  *
  * These 7 methods are a subset of the 25 methods specified in
  * the interface that comes with Java: interface java.util.List<E>
  */
 public interface OurList<E>  {

  // Insert an element at the specified location
  // Precondition: insertIndex >= 0 and insertIndex <= size()
  public void add(int insertIndex, E element) throws IllegalArgumentException;

  // Get the element stored at a specific index
  // Precondition: insertIndex >= 0 and insertIndex < size()
  public E get(int getIndex) throws IllegalArgumentException;
```

```
  // Return the number of elements currently in the list
  public int size();

  // Replace the element at a specific index with element
  // Precondition: insertIndex >= 0 and insertIndex < size()
  public void set(int insertIndex, E element) throws IllegalArgumentException;

  // Return a reference to element in the list or null if not found.
  public E find(E search);

  // Remove element specified by removalIndex if it is in range
  // Precondition: insertIndex >= 0 and insertIndex < size()
  public void removeElementAt(int removalIndex) throws IllegalArgumentException;

  // Remove the first occurrence of element and return true or if the
  // element is not found leave the list unchanged and return false
  public boolean remove(E element);
}
```

OurArrayList<E> implements OurList<E>

The following class implements `OurList` using an array as the structure to store elements. The constructor ensures the array has the capacity to store 10 elements. (The capacity can change). Since `n` is initially set to 0, the list is initially empty.

```
public class OurArrayList<E> implements OurList<E> {

 /**
   * A class constant to set the capacity of the array.
   * The storage capacity increases if needed during an add.
   */
  public static final int INIITIAL_CAPACITY = 10;

 /**
   * A class constant to help control thrashing about when adding and
   * removing elements when the capacity must increase or decrease.
   */
  public static final int GROW_SHRINK_INCREMENT = 10;

  // --Instance variables
  private Object[] data; // Use an array data structure to store elements
  private int n;         // The number of elements (not the capacity)

  /**
   * Construct an empty list with an initial default capacity.
   * This capacity may grow and shrink during add and remove.
   */
  public OurArrayList() {
    data = new Object[INIITIAL_CAPACITY];
    n = 0;
  }
```

Whenever you are making a generic collection, the type parameter (such as <E>) does not appear in the constructor. Since the compiler does not know what the array element type will be in the future, it is declared to be an array of Objects so it can store any reference type.

The initial capacity of any `OurList` object has been selected the be 10 (this is the same as Java's `ArrayList<E>`). This class does not currently have additional constructors to start with a larger capacity, or a different grow and shrink increment, as does Java's `ArrayList.` Enhancing this class in this manner is left as an exercise.

size

The `size` method returns the number of elements in the list which, when empty, is zero.

```
public void testSizeWhenEmpty() {
  OurList<String> emptyList = new OurArrayList<String>();
  assertEquals(0, emptyList.size());
}
```

Because returning an integer does not depend on the number of elements in the collection, the `size` method executes in constant time.

```
/**
  * Accessing method to determine how many elements are in this list.
  * Runtime: O(1)
  * @returns the number of elements in this list.
  */
  public int size() {
    return n;
  }
```

get

OurList specifies a get method that emulates the array square bracket notation `[]` for getting a reference to a specific index. This implementation of the get method will throw an `IllegalArgumentException` if argument `index` is outside the range of `0` through `size()-1`. Although not specified in the interface, this design decision will cause the correct exception to be thrown in the correct place, even if the index is in the capacity bounds of the array. This avoids returning null or other meaningless data during a "get" when the index is in the range of 0 through `data.length-1` inclusive.

```
/**
  * Return a reference to the element at the given index.
  * This method acts like an array with [] except an exception
  * is thrown if index >= size().
  * Runtime: O(1)
  * @returns Reference to object at index if 0 <= index < size().
  * @throws IllegalArgumentException when index<0 or index>=size().
  */
  public E get(int index) throws IllegalArgumentException {
    if (index < 0 || index >= size())
      throw new IllegalArgumentException("" + index);
    return (E)data[index];
  }
```

14.2 Exception Handling

When programs run, errors occur. Perhaps an arithmetic expression results in division by zero, or an array subscript is out of bounds, or to read from a file with a name that simply does not exist. Or perhaps, the `get` method receives an argument 5 when the size was 5. These types of errors that occur while a program is running are known as exceptions.

The `get` method throws an exception to let the programmer using the method know that an invalid argument was passed during a message. At that point, the program terminates indicating the file name, the method name, and the line number where the exception was thrown. When `size` is 5 and the argument 5 is passed, the `get` method throws the exception and Java prints this information:

```
java.lang.IllegalArgumentException: 5
at OurArrayListTest.get(OurArrayListTest.java:108)
```

Programmers have at least two options for dealing with these types of errors:

- Ignore the exception and let the program terminate
- Handle the exception

Java allows you to *try* to execute methods that may throw an exception. The code exists in a `try` block—the keyword `try` followed by the code wrapped in a block, `{ }`.

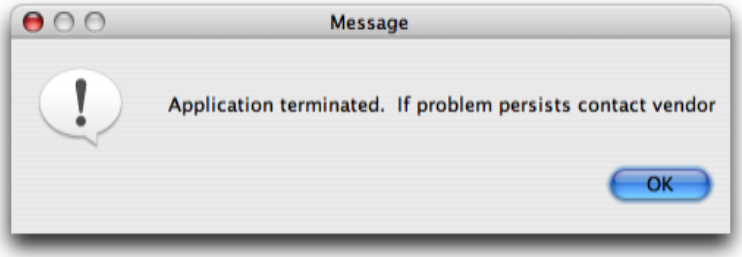
```
try {
    code that may throw an exception when an exception is thrown
}
catch(Exception anException) {
    code that executes only if an exception is thrown from code in the above try block.
}
```

A `try` block must be followed by a `catch` block—the keyword `catch` followed by the anticipated exception as a parameter and code wrapped in a block. The `catch` block contains the code that executes when the code in the `try` block causes an exception to be thrown (or called a method that throws an exception).

Because all exception classes extend the `Exception` class, the type of exception in as the parameter to catch could always be `Exception`. In this case, the `catch` block would catch any type of exception that can be thrown. However, it is recommended that you use the specific exception that is expected to be thrown by the code in the `try` block, such as `IllegalArgumentException`.

The following example will always throw an exception since the list is empty. Any input by the user for `index` will cause the get method to throw an exception.

```
Scanner keyboard = new Scanner(System.in);
OurArrayList<String> list = new OurArrayList<String>();
int index = keyboard.nextInt();

try {
  String str = list.get(index); // When size==0, get throws an exception
}
catch (IllegalArgumentException iobe) {
  JOptionPane.showMessageDialog(null, "Application terminated. "
                          + " If problem persists contact vendor");
}
```



If the size were greater than 0, the user input may or may not cause an exception to be thrown.

To successfully handle exceptions, a programmer must know if a method might throw an exception, and if so, the type of exception. This is done through documentation in the method heading.

```
public E get(int index)throws IllegalArgumentException {
```

A programmer has the option to put a call to get in a `try` block or the programmer may call the method without placing it in a `try` block. The option comes from the fact that `IllegalArgumentException` is a `RuntimeException` that needs not be handled. Exceptions that don't need to be handled are called unchecked exceptions. The unchecked exception classes are those that extend `RuntimeException`, plus any `Exception` that you write that also extends `RuntimeException`. The unchecked exceptions include the following types (this is not a complete list):

- `ArithmeticException`
- `ClassCastException`
- `IllegalArgumentException`
- `IllegalArgumentException`
- `NullPointerException`

Other types of exceptions require that the programmer handle them. These are called checked exceptions. There are many checked exceptions when dealing with file input/output and networking that *must* be surrounded by a try catch. For example when using the `Scanner` class to read input from a file, the constructor needs a `java.io.File` object. Because that constructor can throw a `FileNotFoundException`, the Scanner must be constructed in a try bock.

```
Scanner keyboard = new Scanner(System.in);
String fileName = keyboard.nextLine();
Scanner inputFile = null;

try {
  // Throws exception if file with the input name can not be found
  inputFile = new Scanner(new File(filename));
}
catch (FileNotFoundException fnfe) {
  JOptionPane.showMessageDialog(null, "File not found: '" +
                                      fileName + "'");
}
```

Output assuming the user entered *WrongNameWhoops.data* and that file name does not exist:

Self-Check

14-1 Which of the following code fragments throws an exception?

-a
```
int j = 7 / 0;
```

-b
```
String[] names = new String[5];
names[0] = "Chris";
System.out.println(names[1].toUpperCase());
```

-c
```
String[] names;
names[0] = "Kim";
```

14-2 Write a method that reads and prints all the lines in the file.

Testing that the Method throws the Exception

The `get` method is supposed to throw an exception when the index is out of bounds. To make sure this happens, the following test method will fail if the `get` method does *not* throw an exception when it is expected:

```
@Test
public void testEasyGetException() {
  OurArrayList<String> list = new OurArrayList<String>();
  try {
    list.get(0); // We want get to throw an exception . . .
    fail();      // Show the red bar only if get did NOT throw the exception
  }
  catch (IllegalArgumentException iobe) {
    // . . . and then skip fail() to execute this empty catch block
  }
}
```

This rather elaborate way of testing—to make sure a method throws an exception without shutting down the program—depends on the fact that the empty catch block will execute rather than the `fail` method. The `fail` method of class TestCase automatically generates a failed assertion. The assertion will fail only when your method does not throw an exception at the correct time.

JUnit now provides an easier technique to ensure a method throws an exception. The `@Test` annotation takes a parameter, which can be the type of the Exception that the code in the test method should throw. The following test method will fail if the `get` method does *not* throw an exception when it is expected:

```
@Test(expected = IllegalArgumentException.class)
public void testEasyGetException() {
  OurArrayList<String> list = new OurArrayList<String>();
  list.get(0);   // We want get to ensure this does throws an exception.
}
```

We will use this shorter technique.

add(int, E)

An element of any type can be inserted into any index as long as it in the range of 0 through size() inclusive. Any element added at 0 is the same as adding it as the first element in the list.

```
@Test
public void testAddAndGet() {
  OurList<String> list = new OurArrayList<String>();
  list.add(0, "First");
  list.add(1, "Second");
  list.add(0, "New first");
  assertEquals(3, list.size());
  assertEquals("New first", list.get(0));
  assertEquals("First", list.get(1));
  assertEquals("Second", list.get(2));
}

@Test(expected = IllegalArgumentException.class)
public void testAddThrowsException() {
  OurArrayList<String> list = new OurArrayList<String>();
  list.add(1, "Must start with 0");
}
```

The `add` method first checks to ensure the parameter `insertIndex` is in the correct range. If it is out of range, the method throws an exception.

```
/**
  * Place element at insertIndex.
  * Runtime: O(n)
  *
  * @param element The new element to be added to this list
  * @param insertIndex The location to place the new element.
  * @throws IllegalArgumentException if insertIndex is out of range.
  */
 public void add(int insertIndex, E element) throws IllegalArgumentException {
   // Throw exception if insertIndex is not in range
   if (insertIndex < 0 || insertIndex > size())
     throw new IllegalArgumentException("" + insertIndex);

   // Increase the array capacity if necessary
   if (size() == data.length)
     growArray();

   // Slide all elements right to make a hole at insertIndex
   for (int index = size(); index > insertIndex; index--)
     data[index] = data[index - 1];

   // Insert element into the "hole" and increment n.
   data[insertIndex] = element;
   n++;
 }
```

If the index is in range, the method checks if the array is full. If so, it calls the private helper method `growArray` (shown later) to increase the array capacity. A `for` loop then slides the array elements one index to the right to make a "hole" for the new element. Finally, the reference to the new element gets inserted into the array and n (size) increases by +1. Here is a picture of the array after five elements are added with the following code:

```
OurList<String> list = new OurArrayList<String>();
list.add(0, "A");
list.add(1, "B");
list.add(2, "C");
list.add(3, "D");
list.add(4, "E");
```

n: 5
data:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
"A"	"B"	"C"	"D"	"E"	null	null	null	null	null

At this point, an `add(0, "F")` would cause all array elements to slide to the right by one index. This leaves a "hole" at index 0, which is actually an unnecessary reference to the `String` object `"A"` in `data[0]`. This is okay, since this is precisely where the new element `"F"` should be placed.

n: 5
data:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
"A"	**"A"**	**"B"**	**"C"**	**"D"**	**"E"**	null	null	null	null

After storing a reference to `"F"` in `data[0]` and increasing `n`, the instance variables should look like this:

n: **6**
data:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
"F"	"A"	"B"	"C"	"D"	"E"	null	null	null	null

An `add` operation may require that every reference slide to the bordering array location. If there are 1,000 elements, the loop executes 1,000 assignments. Assuming that an insertion is as likely to be at index 1 as at index n-1 or any other index, the loop will likely average n/2 assignments to make a "hole" for the new element. With the possibility of `growArray` executing O(n), `add,` for all other cases, f(n) = n/2 + n or 1.5n. After dropping the coefficient 1.5, the runtime of `add` would still be O(n). The tightest upper bound is still O(n) even if `growArray` (see below) gets called, since it too is O(n).

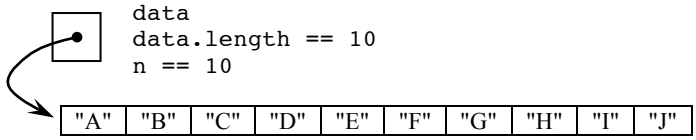
growArray()

The add method checks to see if there is a need to increase the capacity of the array data structure. The add method calls `growArray` which is known as a private helper method, a method that does something useful that is big enough to make it into a method. Because `growAxrray` is inside class `OurArrayList`, `growArray` can be called from anywhere inside the class. And because `data` is an instance variable, any `OurArrayList` object can change `data` to reference a new array with more capacity. This is accomplished with the following algorithm:

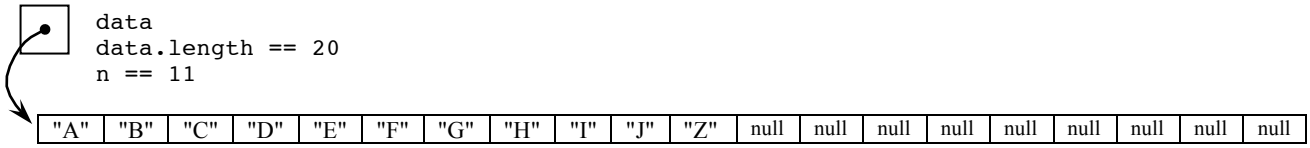
- Make a temporary array that is *`GROW_SHRINK_INCREMENT`* bigger than the instance variable.
- Copy the original contents (`data[0]` through `data[n - 1]`) into this temporary array.
- Assign the reference to the larger array to the array instance variable

```
// Change data to have the same elements in indexes 0..n - 1
// and have the same number of new array locations to store new elements.
private void growArray() {
  String[] temp = new String[n + GROW_SHRINK_INCREMENT];
  // Copy all existing elements into the new and larger array
  for (int index = 0; index < n; index++) {
    temp[index] = data[index];
  }
  // Store a reference to the new bigger array as part of this object's state
  data = temp;
}
```

When the array is filled to capacity (with the Strings `"A"` through `"J"` added in this example), the instance variables `data` and `n` look like this:



During the message `add("Z")`; the `add` method would send the `growArray` message in order to increase the capacity by 10. The instance variables would change to this picture of memory:



Note: The `growArray` method is declare private because it is better design to *not* clutter the public part of a class with things that users of the class are not able to use or are not interested in using. It is good practice to hide details from users of your software.

find(E)

The `find` method returns a reference to the first element in the list that matches the argument. It uses the `equals` method that is defined for that type. The `find` method returns `null` if the argument does not match any list element, again using the `equals` method for the type of elements being stored. Any class of objects that you store should override the `equals` method such that the state of the objects are compared rather than references.

Searching for a `String` in a list of strings is easy, since the `String` equals method does compare the state of the object. You can simply ask to get a reference to a `String` by supplying the string you seek as an argument.

```
@Test
public void testFindWithStrings() {
  OurList<String> list = new OurArrayList<String>();
  list.add(0, "zero");
  list.add(1, "one");
  list.add(2, "two");
  assertNotNull(list.find("zero"));
  assertNotNull(list.find("one"));
  assertNotNull(list.find("two"));
}
```

A test should also exist to make sure `null` is returned when the string does not exist in the list

```
@Test
public void testFindWhenNotHere() {
  OurList<String> names = new OurArrayList<String>();
  names.add(0, "Jody");
  names.add(1, "Devon");
  names.add(2, "Nar");
  assertNull(names.find("Not Here"));
}
```

However, for most other types, searching through an `OurArrayList` object (or an `ArrayList` or `LinkedList` object) requires the creation of a faked temporary object that "`equals`" the object that you wish to query or whose state you wish to modify. Consider the following test that establishes a small list for demonstration purposes. Using a small list of `BankAccount` objects, the following code shows a deposit of 100.00 made to one of the accounts.

```
@Test
public void testDepositInList() {
  OurList<BankAccount> accountList = new OurArrayList<BankAccount>();
  accountList.add(0, new BankAccount("Joe", 0.00));
  accountList.add(1, new BankAccount("Ali", 1.00));
  accountList.add(2, new BankAccount("Sandeep", 2.00));

  String searchID = "Ali";
  // Make an account that EQUALS an element in the array (ID only needed
  BankAccount searchAccount = new BankAccount(searchID, -999);
  BankAccount ref = accountList.find(searchAccount);
  ref.deposit(100.00);
  // Make sure the correct element was really changed
  ref = accountList.find(searchAccount);
  assertEquals(101.00, ref.getBalance(), 1e-12);
}
```

The code constructs a "fake" reference (`searchAccount`) to be compared to elements in the list. This temporary instance of `BankAcccount` exists solely for aiding the search process. To make an appropriate temporary search object, the programmer must know how the equals method returns true for this type when the IDs match exactly. (You may need to consult the documentation for `equals` methods of other type.) The temporary search account need only have the ID of the searched-for object—`equals` ignores the balance. They do not need to match the other parts of the real object's state.[3] The constructor uses an initial balance of -999 to emphasize that the other parameter will not be used in the search.

The `find` method uses the sequential search algorithm to search the unsorted elements in the array structure. Therefore it runs O(n).

```
/** Return a reference to target if target "equals" an element in this list,
  * or null if not found.   Runtime: O(n)
  * @param target The object that will be compared to list elements.
  * @returns Reference to first object that equals target (or null).
  */
  public E find(E target) {
    // Get index of first element that equals target
    for (int index = 0; index < n; index++)  {
      if (target.equals(data[index]))
        return data[index];
    }
    return null; // Did not find target in this list
  }
```

The following test method builds a list of two `BankAccount` objects and asserts that both can be successfully found.

```
@Test
public void testFindWithBankAccounts() {
  // Set up a small list of BankAccounts
  OurList<BankAccount> list = new OurArrayList<BankAccount>();
  list.add(0, new BankAccount("zero", 0.00));
  list.add(1, new BankAccount("one", 1.00));

  // Find one
  BankAccount fakedToFind = new BankAccount("zero", -999);
  BankAccount withTheRealBalance = list.find(fakedToFind);

  // The following assertions expect a reference to the real account
  assertNotNull(withTheRealBalance);
  assertEquals("zero", withTheRealBalance.getID());
  assertEquals(0.00, withTheRealBalance.getBalance(), 1e-12);

  // Find the the other
  fakedToFind = new BankAccount("one", +234321.99);
  withTheRealBalance = list.find(fakedToFind);

  // The following assertions expect a reference to the real account
  assertNotNull(withTheRealBalance);
  assertEquals("one", withTheRealBalance.getID());
  assertEquals(1.00, withTheRealBalance.getBalance(), 1e-12);
}
```

[3] This is typical when searching through indexed collections. However, there are better ways to do the same thing. Other collections map a key to a value. All the programmer needs to worry about is the key, such as an account number or student ID. There is no need to construct a temporary object or worry about how a particular `equals` method works for many different classes of objects.

And of course we should make sure the `find` method returns null when the object does not "equals" any element in the list:

```
@Test
  public void testFindWhenElementIsNotThere() {
    OurList<BankAccount> list = new OurArrayList<BankAccount>();
    list.add(0, new BankAccount("zero", 0.00));
    list.add(1, new BankAccount("one", 1.00));
    list.add(2, new BankAccount("two", 2.00));
    BankAccount fakedToFind = new BankAccount("Not Here", 0.00);
    // The following assertions expect a reference to the real account
    assertNull(list.find(fakedToFind));
  }
```

The other methods of `OurList` are left as an optional exercise.

Answers to Self-Check Questions

14-1 which throws an exception? a, b, and c

-a- / by 0

-b- NullPointerException because `names[1]` is `null`

-c- No runtime error, but this is a compiletime error because names is not initialized

14-2 read in from a file

```
public void readAndPrint(String fileName) {
  Scanner inputFile = null; // To avoid compiletime error later
  try {
    inputFile = new Scanner(new File(fileName));
  }
  catch (FileNotFoundException e) {
    e.printStackTrace();
  }

  while (inputFile.hasNextLine()) {
     System.out.println(inputFile.nextLine());
  }
}
```

Chapter 15

Linked Structures

This chapter demonstrates the `OurList` interface implemented with a class that uses a linked structure rather than the array implementation of the previous chapter. The linked version introduces another data structure for implementing collection classes: the singly linked structure.

Goals

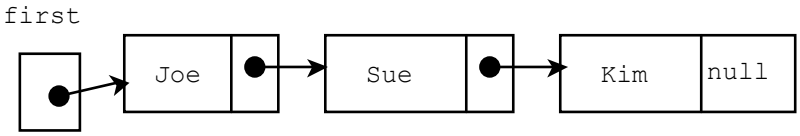
- Show a different way to elements in a collection class
- See how nodes can be linked
- Consider the differences from arrays in order to such as sequencing through elements that are no longer in contiguous memory

---

15.1 The Singly Linked Structure

A collection of elements can be stored within a linked structure by storing a reference to elements in a node and a reference to another node of the same type. The next node in a linked structure also stores a reference to data and a reference to yet another node. There is at least one variable to locate the beginning, which is named `first` here.

A linked structure with three nodes



Each node is an object with two instance variables:

1. A reference to the data of the current node ("Joe", "Sue", and "Kim" above)
2. A reference to the next element in the collection, indicated by the arrows

The node with the reference value of `null` indicates the end of the linked structure. Because there is precisely one link from every node, this is a singly linked structure. (Other linked structures have more than one link to other nodes.)

A search for an element begins at the node referenced by the external reference `first`. The second node can be reached through the link from the first node. Any node can be referenced in this sequential fashion. The search stops at the null terminator, which indicates the end. These nodes may be located anywhere in available memory. The elements are not necessarily

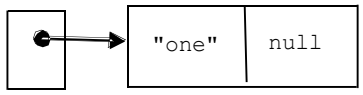
contiguous memory locations as with arrays. Interface `OurList` will now be implemented using many instances of the private inner class named `Node`.

```
/**
 * OurLinkedList is a class that uses an singly linked structure to
 * store a collection of elements as a list. This is a growable coll-
 * ection that uses a linked structure for the backing data storage.
 */
public class OurLinkedList<E> implements OurList<E> {
  // This private inner class is accessible only within OurLinkedList.
  // Instances of class Node will store a reference to the same
  // type used to construct an OurLinkedList<Type>.
  private class Node  {
    // These instance variables can be accessed within OurLinkedList<E>
    private E data;
    private Node next;

    public Node(E element) {
      data = element;
      next = null;
    }
  } // end class Node

  // TBA: OurLinkedList instance variables and methods

} // end class OurLinkedList
```

The `Node` instance variable `data` is declared as Object in order to allow any type of element to be stored in a node. The instance variable named `next` is of type `Node`. This allows one `Node` object to refer to another instance of the same `Node` class. Both of these instance variables will be accessible from the methods of the enclosing class (`OurLinkedList`) even though they have `private` access.

We will now build a linked structure storing a collection of three `String` objects. We let the `Node` reference `first` store a reference to a Node with `"one"` as the data.

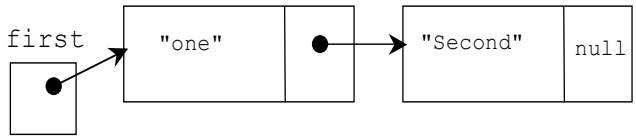
```
  // Build the first node and keep a reference to this first node
  Node first = new Node("one");
```




```
public Node(Object objectReference) {
  data = objectReference;
  next = null;
}
```

The following construction stores a reference to the string `"second"`. However, this time, the reference to the new `Node` object is stored into the `next` field of the `Node` referenced by `first`. This effectively adds a new node to the end of the list.

```
 // Construct a second node referenced by the first node's next
 first.next = new Node("Second");
```



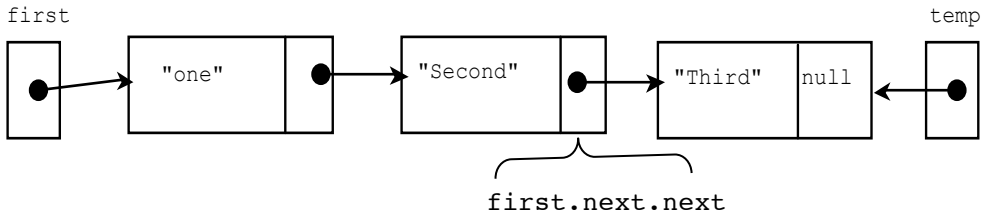
The code above directly assigned a reference to the next instance variable. This unusual direct reference to a private instance variable makes the implementation of `OurLinkedList` than having a separate class as some textbooks use. Since `Node` is intimately tied to this linked

structure — and it has been made an inner class — you will see many permitted assignments to both of `Node`'s private instance variables, `data` and `next`.

This third construction adds a new `Node` to the end of the list. The `next` field is set to refer to this new node by referencing it with the dot notation `first.next.next`.

```
// Construct a third node referenced by the second node's next
Node temp = new Node("Third");
// Replace null with the reference value of temp (pictured as an arrow)
first.next.next = temp;
```

The following picture represents this hard coded (not general) list:

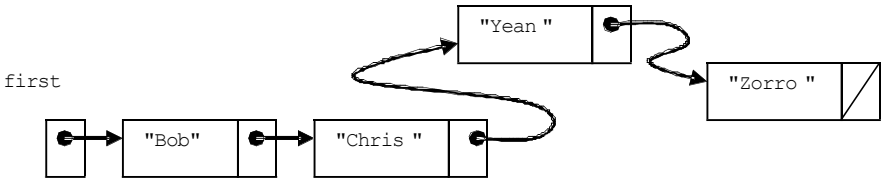


The `Node` reference variable named `first` is not an internal part of the linked structure. The purpose of `first` is to find the beginning of the list so algorithms can find an insertion point for a new element, for example.

In a singly linked structure, the instance variable `data` of each `Node` refers to an object in memory, which could be of any type. The instance variable `next` of each `Node` object references another node containing the next element in the collection. One of these `Node` objects stores `null` to mark the end of the linked structure. The `null` value should only exist in the last node.

Self-Check

Use this linked structure to answer the questions that follow.



15-1 What is the value of `first.data`?

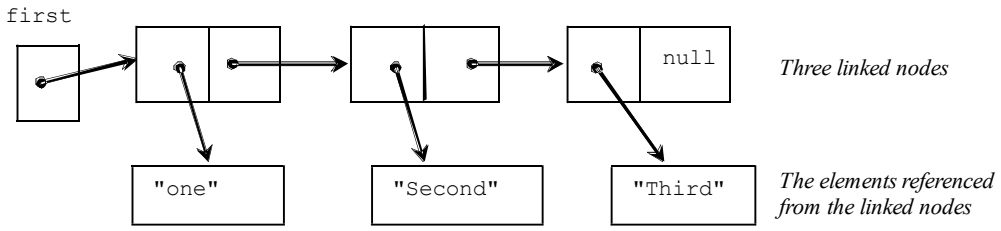
15-2 What is the value of `first.next.data`?

15-3 What is the value of `first.next.next.next.data`?

15-4 What is the value of `first.next.next.next`?

Each node stores a reference to the element

A linked structure would be pictured more accurately with the `data` field shown to reference an object somewhere else in memory.



However, it is more convenient to show linked structures with the value of the element written in the node, especially if the elements are strings. This means that even though both parts store a reference value (exactly four bytes of memory to indicate a reference to an object), these structures are often pictured with a box dedicated to the data value, as will be done in the remainder of this chapter. The reference values, pictured as arrows, are important. If one of these links becomes misdirected, the program will not be able to find elements in the list.

Traversing a Linked Structure

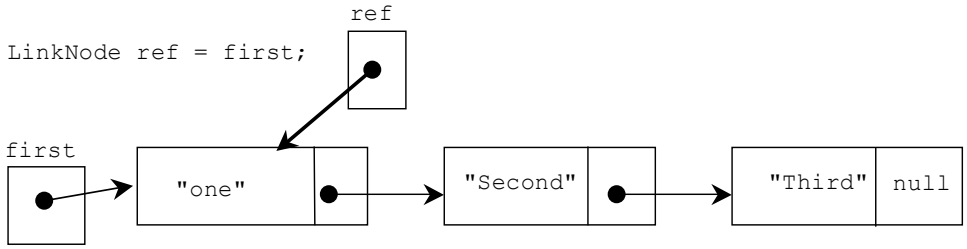
Elements in a linked structure can be accessed in a sequential manner. Analogous to a changing `int` subscript to reference all elements in an array, a changing `Node` variable can reference all elements in a singly linked structure. In the following `for` loop, the `Node` reference begins at the first node and is updated with `next` until it becomes `null`.

```
for (Node ref = first; ref != null; ref = ref.next) {
  System.out.println(ref.data.toString());
```

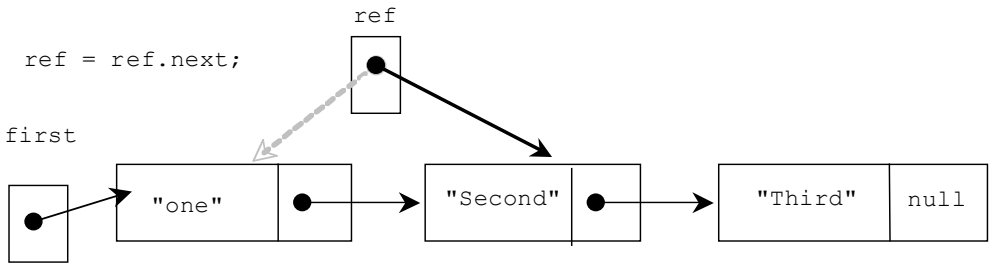
Output

```
one
Second
Third
```

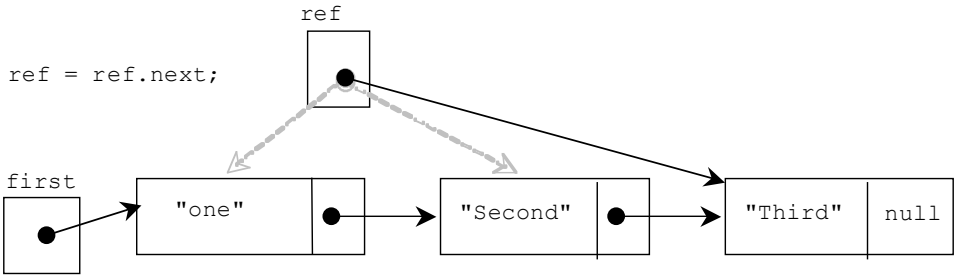
When the loop begins, `first` is not `null`, thus neither is `ref`. The `Node` object `ref` refers to the first node.



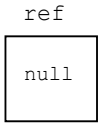
At this point, `ref.data` returns a reference to the object referenced by the data field—in this case, the string "one". To get to the next element, the `for` loop updates the external pointer `ref` to refer to the next node in the linked structure. The first assignment of `ref = ref.next` sets `ref` to reference the second node.



At the end of the next loop iteration, `ref = ref.next` sets `ref` to reference the third node.



And after one more `ref = ref.next`, the external reference named `ref` is assigned `null`.



At this point, the `for` loop test `ref != null` is `false`. The traversal over this linked structure is complete.

With an array, the `for` loop could be updating an integer subscript until the value is beyond the index of the last meaningful element (`index == n` for example). With a linked structure, the `for` loop updates an external reference (a `Node` reference named `ref` here) so it can reference all nodes until it finds the `next` field to be `null`.

This traversal represents a major difference between a linked structure and an array. With an array, subscript `[2]` directly references the third element. This random access is very quick and it takes just one step. With a linked structure, you must often use sequential access by beginning at the first element and visiting all the nodes that precede the element you are trying to access. This can make a difference in the runtime of certain algorithms and drive the decision of which storage mechanism to use.

15.2 Implement `OurList` with a Linked Structure

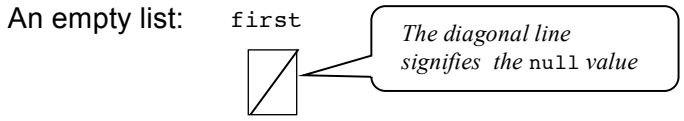
Now that the inner `private class Node` exists, consider a class that implements `OurList`. This class will provide the same functionality as `OurArrayList` with a different data structure. The storage mechanism will be a collection of `Node` objects. The algorithms will change to accommodate this new underlying storage structure known as a singly linked structure. The collection class that implements ADT `OurList` along with its methods and linked structure is known as a **linked list**.

This `OurLinkedList` class uses an inner Node class with two additional constructors (their use will be shown later). It also needs the instance variable first to mark the beginning of the linked structure.

```
// A type-safe Collection class to store a list of any type element
public class OurLinkedList<E> implements OurList<E> {

  // This private inner class is only known within OurLinkedList.
  // Instances of class Node will store a reference to an
  // element and a reference to another instance of Node.
  private class Node  {

    // Store one element of the type specified during construction
    private E data;
    // Store a reference to another node with the same type of data
    private Node next;


    // Allows Node n = new Node();
    public Node() {
      data = null;
      next = null;
    }

    // Allows Node n = new Node("Element");
    public Node(E element) {
      data = element;
      next = null;
    }

    // Allows Node n = new Node("Element", first);
    public Node(E element, Node nextReference) {
      data = element;
      next = nextReference;
    }

  } /////// end inner class Node //////////

  // Instance variables for OurLinkedList
  private Node first;

  private int size;

  // Construct an empty list with size 0
  public OurLinkedList() {
    first = null;
    size = 0;
  }

  // more to come …
}
```

After construction, the picture of memory shows `first` with a `null` value written as a diagonal line.

```
OurLinkedList<String> list = new OurLinkedList<String>();
```



The first method `isEmpty` returns true when `first` is `null`.

```
/**
 * Return true when no elements are in this list
 */
public boolean isEmpty() {
  return first == null;
}
```

Adding Elements to a Linked Structure

This section explores the algorithms to add to a linked structure in the following ways:

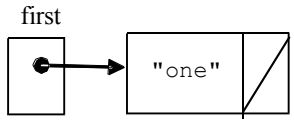
- Inserting an element at the beginning of the list
- Inserting an element at the end of the list
- Inserting an element anywhere in the list at a given position

To insert an element as the first element in a linked list that uses an external reference to the first node, the algorithm distinguishes these two possibilities:

1. the list is empty
2. the list is not empty

If the list is empty, the insertion is relatively easy. Consider the following code that inserts a new `String` object at index zero of an empty list. A reference to the new `Node` object with `"one"` will be assigned to `first`.

```
OurLinkedList<String> stringList = new OurLinkedList<String>();
stringList.addFirst("one");
```




```
/** Add an element to the beginning of this list.
  * O(1)
  * @param element The new element to be added at index 0.
  */
  public void addFirst(E element) {
    // The beginning of an addFirst operation
    if (this.isEmpty()) {
      first = new Node(element);
      // ...
```

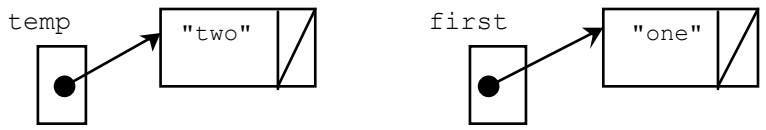
When the list is not empty, the algorithm must still make the insertion at the beginning; `first` must still refer to the new first element. You must also take care of linking the new element to the rest of the list. Otherwise, you lose the entire list! Consider adding a new first element (to a list that is not empty) with this message:

```
stringList.addFirst("two");
```

This can be accomplished by constructing a new `Node` with the zero-argument constructor that sets both `data` and `next` to `null`. Then the reference to the soon to be added element is stored in the data field (again E can represent any reference type).

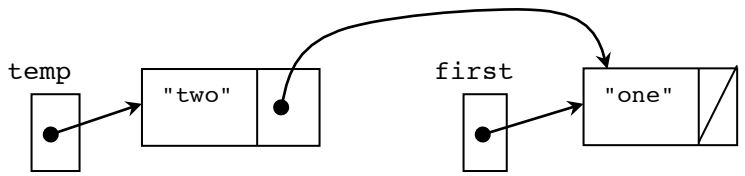
```
    else {
      // the list is NOT empty
      Node temp = new Node(); // data and next are null
      temp.data = element;    // Store reference to element
```

There are two lists now, one of which is temporary.



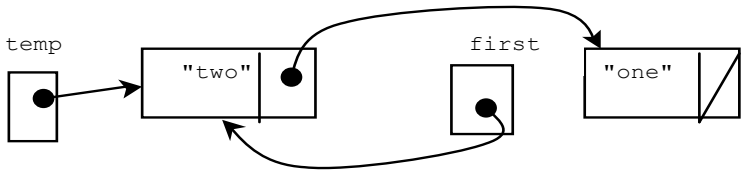
The following code links the node that is about to become the new `first` so that it refers to the element that is about to become the second element in the linked structure.

```
    temp.next = first; // 2 Nodes refer to the node with "one"
  }
```

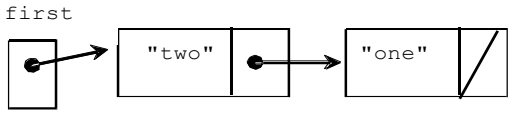


Now move `first` to refer to the `Node` object referenced by `first` and increment `size`.

```
      first = temp;
    }  // end method addFirst
    size++;
  } // end addFirst
```



After `"two"` is inserted at the front, the local variable `temp` is no longer needed in the picture. The list can also be drawn like this since the local variable `temp` will no longer exist after the method finishes:



This `size` method can now return the number of elements in the collection (providing the other `add` methods also increment size when appropriate).

```
  /**
    * Return the number of elements in this list
    */
   public int size() {
     return size;
   }
```

Self-Check

15-5 Draw a picture of memory after each of the following sets of code executes:

a.
```
OurLinkedList<String> aList = new OurLinkedList<String>();
```

b.
```
OurLinkedList<String> aList = new OurLinkedList<String>();
aList.addFirst("Chris");
```

c.
```
OurLinkedList<Integer> aList = new OurLinkedList<Integer>();
aList.addFirst(1);
aList.addFirst(2);
```

`addFirst(E)` again

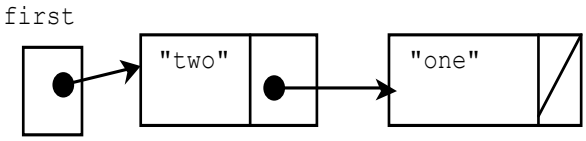
The `addFirst` method used an `if…else` statement to check if the reference to the beginning of the list needed to be changed. Then several other steps were required. Now imagine changing the `addFirst` method using the two-argument constructor of the `Node` class.

```
public Node(Object element, Node nextReference) {
  data = element;
  next = nextReference;
}
```

To add a new node at the beginning of the linked structure, you can initialize both `Node` instance variables. This new two-argument constructor takes a `Node` reference as a second argument. The current value of `first` is stored into the new `Node`'s `next` field. The new node being added at index 0 now links to the old `first`.

```
/** Add an element to the beginning of this list.
  * @param element The new element to be added at the front.
  * Runtime O(1)
  */
 public void addFirst(E element) {
   first = new Node(element, first);
   size++;
 }
```

To illustrate, consider the following message to add a third element to the existing list of two nodes (with "two" and "one"): `stringList.addFirst("tre");`



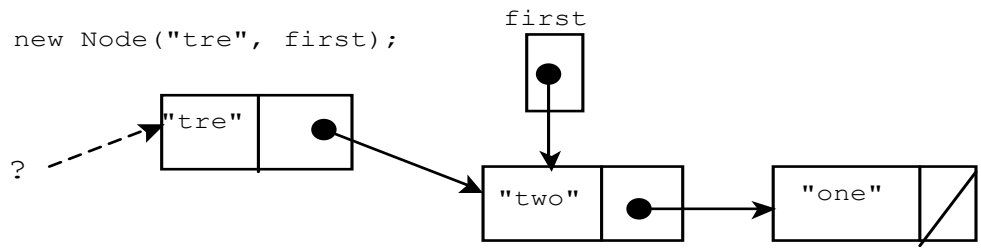
The following initialization executes in `addFirst`:

```
first = new Node(element, first);
```

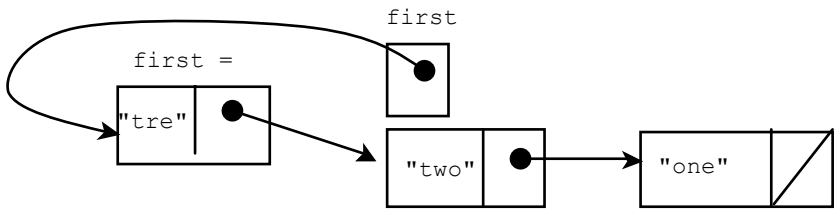
This invokes the two-argument constructor of `class Node`:

```
public Node(Object element, Node nextReference) {
  data = element;
  next = nextReference;
}
```

This constructor generates the `Node` object pictured below with a reference to `"tre"` in `data`. It also stores the value of `first` in its `next` field. This means the new node (with `"tre"`) has its `next` field refer to the old `first` of the list.

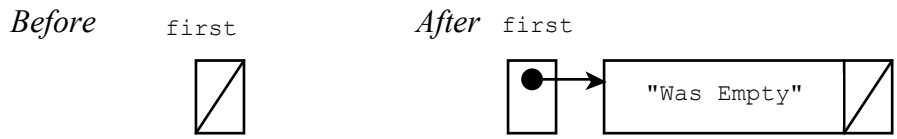


Then after the construction is finished, the reference to the new `Node` is assigned to `first`. Now `first` refers to the new `Node` object. The element `"tre"` is now at index 0.



The following code illustrates that `addFirst` will work even when the list is empty. You end up with a new `Node` whose reference instance variable next has been set to `null` and where `first` references the only element in the list.

```
OurLinkedList<String> anotherList = new OurLinkedList<String>();
anotherList.addFirst("Was Empty");
```



Since the `addFirst` method essentially boils down to two assignment statements, no matter how large the list, `addFirst` is O(1).

E get(int)

OurList specifies a get method that emulates the array square bracket notation `[ ]` for getting a reference to a specific index. This implementation of the `get` method will throw an `IllegalArgumentException` if the argument `index` is outside the range of `0` through `size()-1`. This avoids returning `null` or other meaningless data during a `get` when the index is out of range.

```
/**
  * Return a reference to the element at index getIndex O(n)
  */
 public E get(int getIndex) {
 // Verify insertIndex is in the range of 0..size()-1
 if (getIndex < 0 || getIndex >= this.size())
   throw new IllegalArgumentException("" + getIndex);
```

Finding the correct node is not the direct access available to an array. A loop must iterate through the linked structure.

```
    Node ref = first;
    for (int i = 0; i < getIndex; i++)
      ref = ref.next;
    return ref.data;
  } // end get
```

When the temporary external reference `ref` points to the correct element, the data of that node will be returned. It is now possible to test the `addFirst` and `get` methods. First, let's make sure the method throws an exception when the index to `get` is out of range. First we'll try `get(0)` on an empty list.

```
  @Test(expected = IllegalArgumentException.class)
  public void testGetExceptionWhenEmpty() {
    OurLinkedList<String> list = new OurLinkedList<String>();
    list.get(0); // We want get(0) to throw an exception
  }
```

Another test method ensures that the indexes just out of range do indeed throw exceptions.

```
  @Test(expected = IllegalArgumentException.class)
  public void testGetExceptionWhenIndexTooBig() {
    OurLinkedList<String> list = new OurLinkedList<String>();
    list.addFirst("B");
    list.addFirst("A");
    list.get(2); // should throw an exception
  }

  @Test(expected = IllegalArgumentException.class)
  public void testGetExceptionWhenIndexTooSmall () {
    OurLinkedList<String> list = new OurLinkedList<String>();
    list.addFirst("B");
    list.addFirst("A");
    list.get(-1); // should throw an exception
  }
```

This test for `addFirst` will help to verify it works correctly while documenting the desired behavior.

```
  @Test
  public void testAddFirstAndGet() {
    OurLinkedList<String> list = new OurLinkedList<String>();
    list.addFirst("A");
    list.addFirst("B");
    list.addFirst("C");
    // Assert that all three can be retrieved from the expected index
    assertEquals("C", list.get(0));
    assertEquals("B", list.get(1));
    assertEquals("A", list.get(2));
  }
```

Self-Check

15-6 Which one of the following assertions would fail: a, b, c, or d?

```
OurLinkedList<String> list = new OurLinkedList<String>();
list.addFirst("El");
list.addFirst("Li");
list.addFirst("Jo");
list.addFirst("Cy");
```

```
assertEquals("El", list.get(3));  // a.
assertEquals("Li", list.get(2));  // b.
assertEquals("JO", list.get(1));  // c.
assertEquals("Cy", list.get(0));  // d.
```

String toString()

Programmers using an `OurLinkedList` object may be interested in getting a peek at the current state of the list or finding an element in the list. To do this, the list will also have to be traversed.

This algorithm in `toString` begins by storing a reference to the first node in the list and updating it until it reaches the desired location. A complete traversal begins at the node reference by first and ends at the last node (where the `next` field is `null`). The loop traverses the list until `ref` becomes `null`. This is the only `null` value stored in a next field in any proper list. The null value denotes the end of the list.

```
/**
  * Return a string with all elements in this list.
  * @returns One String that concatenation of toString versions of all
  * elements in this list separated by ", " and bracketed with "[  ]".
  */
  public String toString() {
    String result = "[";
    if (! this.isEmpty()) { // There is at least one element
      // Concatenate all elements except the last
      Node ref = first;
      while (ref.next != null) {
        // Concatenate the toString version of each element
        result = result + ref.data.toString() + ", ";
        // Bring loop closer to termination
        ref = ref.next;
      }
      // Concatenate the last element (if size > 0) but without ", "
      result += ref.data.toString();
    }
    // Always concatenate the closing square bracket
    result += "]";
    return result;
  }
```

Notice that each time through the `while` loop, the variable `ref` changes to reference the `next` element. The loop keeps going as long as `ref` does not refer to the last element (`ref.next != null`) .

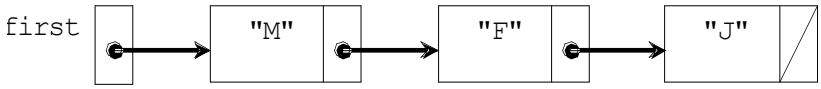
Modified versions of the `for` loop traversal will be used to insert an element into a linked list at a specific index, to find a specific element, and to remove elements.

The add(int, E) Method

Suppose a linked list has the three strings `"M"`, `"F"`, and `"J"`:

```
OurLinkedList<String> list = new OurLinkedList<String>();
list.add(0, "M");
list.add(1, "F");
list.add(2, "J");
assertEquals("[M, F, J]", list.toString());
```

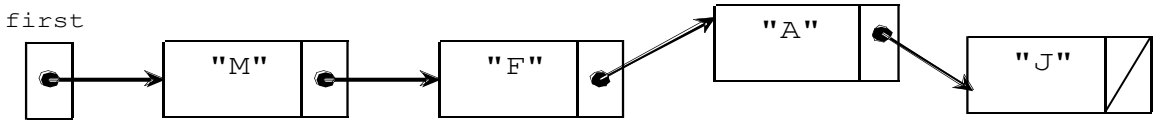
The linked structure generated by the code above would look like this:



This message inserts a fourth string into the 3rd position, at index 2, where "J" is now:

```
list.add(2, "A"); // This has zero based indexing--index 2 is 3rd spot
assertEquals("[M, F, A, J]", list.toString());
```

Since the three existing nodes do not necessarily occupy contiguous memory locations in this list, the elements in the existing nodes need not be shifted as did the array data structure. However, you will need a loop to count to the insertion point. Once again, the algorithm will require a careful adjustment of links in order to insert a new element. Below, we will see how to insert "A" at index 2.



The following algorithm inserts an element into a specific location in a linked list. After ensuring that the index is in the correct range, the algorithm checks for the special case of inserting at index 0, where the external reference `first` must be adjusted.

```
if the index is out of range
    throw an exception
else if the new element is to be inserted at index 0
  addFirst(element)
else {
  Find the place in the list to insert
  construct a new node with the new element in it
  adjust references of existing Node objects to accommodate the insertion
}
```

This algorithm is implemented as the `add` method with two arguments. It requires the index where that new element is to be inserted along with the object to be inserted. If either one of the following conditions exist, the index is out of range:

1. a negative index
2. an index greater than the size() of the list

The `add` method first checks if it is appropriate to throw an exception — when `insertIndex` is out of range.

```
/** Place element at the insertIndex specified.
  * Runtime: O(n)
  * @param element The new element to be added to this list
  * @param insertIndex The location where the new element will be added
  * @throws IllegalArgumentException if insertIndex is out of range
  */
  public void add(int insertIndex, E element) {
    // Verify insertIndex is in the range of 0..size()-1
    if (insertIndex < 0 || insertIndex > this.size())
      throw new IllegalArgumentException("" + insertIndex);
```

The method throws an `IllegalArgumentException` if the argument is less than zero or greater than the number of elements. For example, when the size of the list is 4, the only legal arguments would be 0, 1, 2, or 3 and 4 (inserts at the end of the list). For example, the following message generates an exception because the largest index allowed with "insert element at" in a list of four elements is 4.

```
list.add(5, "Y");
java.lang.IllegalArgumentException: 5
```

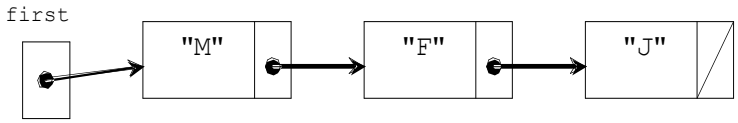
If `insertIndex` is in range, the special case to watch for is if the `insertAtIndex` equals 0. This is the one case when the external reference `first` must be adjusted.

```
if (insertIndex == 0) {
  // Special case of inserting before the first element.
  addFirst(element);
}
```

The instance variable `first` must be changed if the new element is to be inserted before all other elements. It is not enough to simply change the local variables. The `addFirst` method shown earlier conveniently takes the correct steps when `insertIndex==0`.

If the `insertIndex` is in range, but not 0, the method proceeds to find the correct insertion point. Let's return to a list with three elements, built with these messages:

```
OurLinkedList<String> list = new OurLinkedList<String>();
list.add(0, "M");
list.add(1, "F");
list.add(2, "J");
```



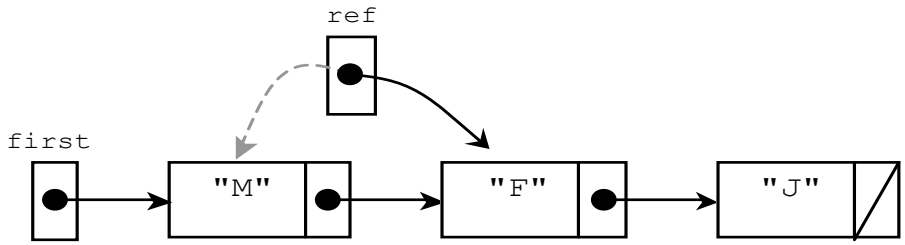
This message inserts a new element at index 2, after `"F"` and before `"J"`.

```
list.add(2, "A"); // We're using zero based indexing, so 2 is 3rd spot
```

This message causes the `Node` variable `ref` (short for reference) to start at `first` and get. This external reference gets updated in the `for` loop.

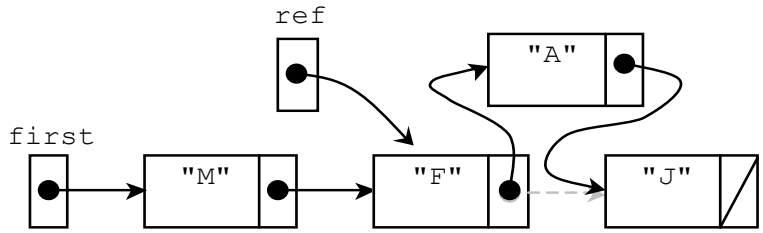
```
else {
  Node ref = first;
  for (int index = 1; index < insertIndex; index++) {
    // Stop when ref refers to the node before the insertion point
    ref = ref.next;
  }
  ref.next = new Node(element, ref.next);
}
```

The loop leaves `ref` referring to the node before the node where the new element is to be inserted. Since this is a singly linked list (and can only go forward from `first` to back) there is no way of going in the opposite direction once the insertion point is passed. So, the loop must stop when ref refers to the node *before* the insertion point.



The insertion can now be made with the `Node` constructor that takes a `Node` reference as a second argument.

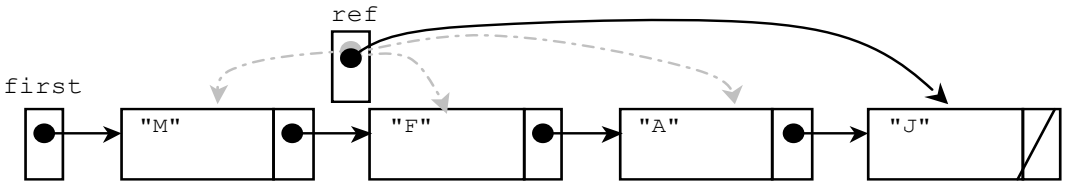
```
ref.next = new Node(element, ref.next);
```



Inserting at index 2

Consider the insertion of an element at the end of a linked list. The `for` loop advances ref until it refers to the last node in the list, which currently has the element "J". The following picture provides a trace of ref using this message

```
list.add(list.size(), "LAST");
```

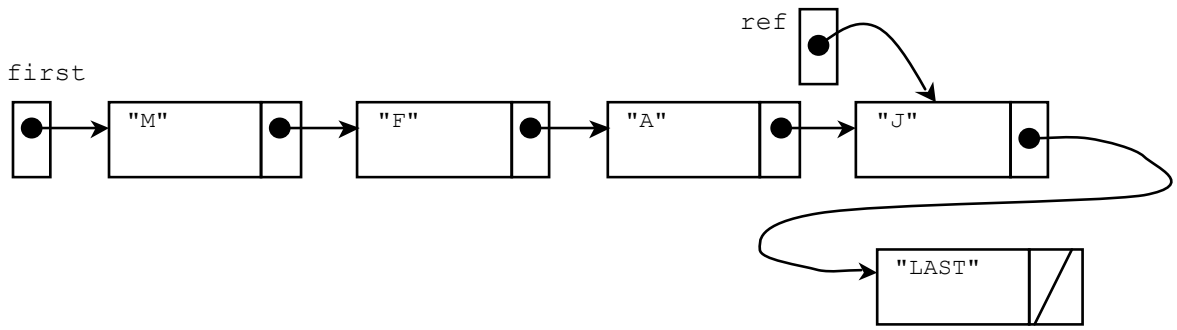


If the list has 1,000 elements, this loop requires 999 (or in general n-1) operations.

```
for (int index = 1; index < insertIndex - 1; index++) {
  ref = ref.next;
}
```

Once the insertion point is found, with `ref` pointing to the correct node, the new element can be added with one assignment and help from the `Node` class.

```
ref.next = new Node(element, ref.next);
```



The new node's `next` field becomes `null` in the `Node` constructor. This new node, with `"LAST"` in it, marks the new end of this list.

Self-Check

15-7 Which of the `add` messages (there may be more than one) would throw an exception when sent immediately after the message `list.add(0, 4);`?

```
OurLinkedList<Integer> list = new OurLinkedList<Integer> ();
list.add(0, 1);
list.add(0, 2);
list.add(0, 3);
list.add(0, 4);
```

a. `list.add(-1, 5);`
b. `list.add(3, 5);`
c. `list.add(5, 5);`
d. `list.add(4, 5);`

addLast

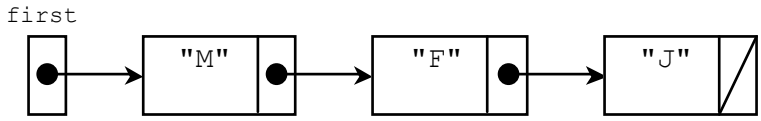
The `addLast` method is easily implemented in terms of `add`. It could have implemented the same algorithm separately, however it is considered good design to avoid repeated code and use an existing method if possible.

```
/**
  * Add an element to the end of this list.
  * Runtime: O(n)
  * @param element The element to be added as the new end of the list.
  */
  public void addLast(E element) {
    // This requires n iterations to determine the size before the
    // add method loops size times to get a reference to the last
    // element in the list. This is n + n operations, which is O(n).
    add(size(), element);
  }
```

The `addLast` algorithm can be modified to run O(1) by adding an instance variable that maintains an external reference to the last node in the linked list. Modifying this method is left as a programming exercise.

Removing from a Specific Location: **`removeElementAt(int)`**

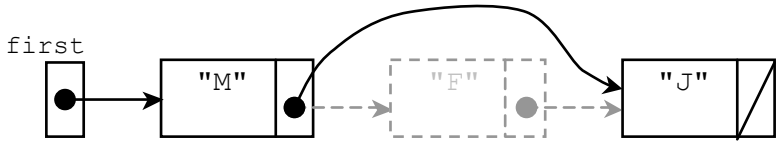
Suppose a linked list has these three elements:



Removing the element at index 1 is done with a `removeElementAt` message.

```
assertEquals("[M, F, J]", list.toString());
list.removeElementAt(1);
assertEquals("[M, J]", list.toString());
```

The linked list should look like this after the node with "F" is reclaimed by Java's garbage collector. There are no more references to this node, so it is no longer needed.

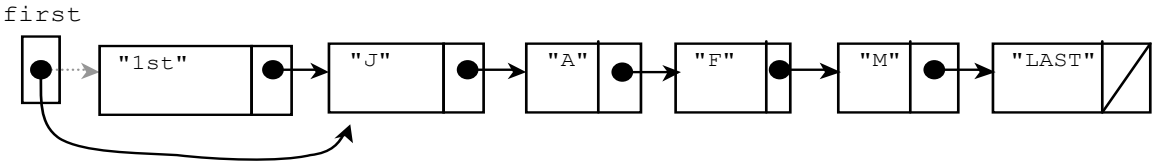


Assuming the index of the element to be removed is in the correct range of `0` through `size()-1`, the following algorithm should work with the current implementation:

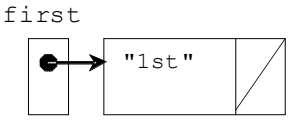
if removal index is out of range
  throw an exception
else if the removal is the node at the `first`
  change `first` to refer to the second element (or make the list empty if size()==1)
else {
  Get a reference to the node before the node to be removed
  Send the link around the node to be removed
}

A check is first made to avoid removing elements that do not exist, including removing index 0 from an empty list. Next up is checking for the special case of removing the first node at index 0 ("one" in the structure below). Simply send `first.next` "around" the first element so it references the second element. The following assignment updates the external reference first to refer to the next element.

```
first = first.next;
```

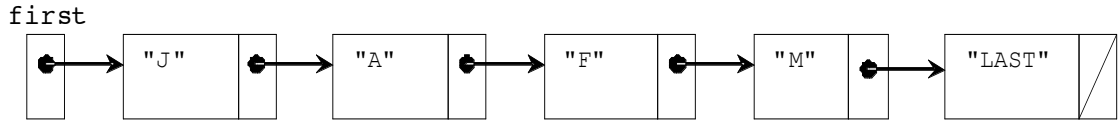


This same assignment will also work when there is only one element in the list.



With the message `list.removeElementAt(0)` on a list of size 1, the old value of first is replaced with `null`, making this an empty list.

Now consider `list.removeElementAt(2)` (`"F"`) from the following list:

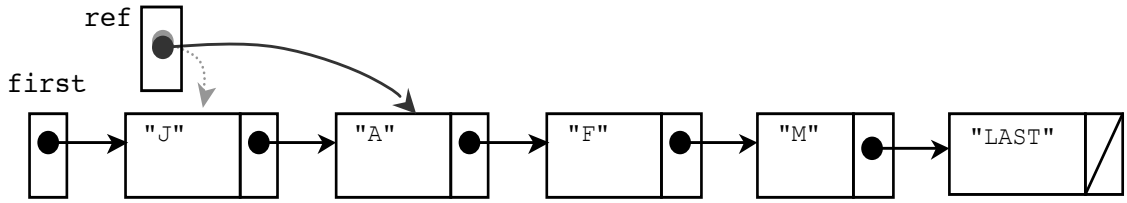


The following assignment has the `Node` variable `ref` refer to the same node as `first`:

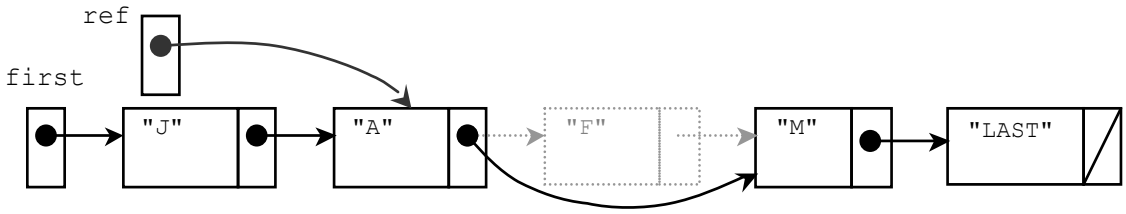
```
Node ref = first;
```

`ref` then advances to refer to the node just *before* the node to be removed.

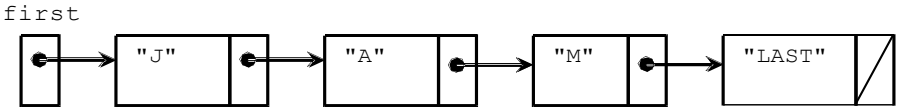
```
for (int index = 1; index < removalIndex; index++) // 1 iteration only
  ref = ref.next;
```



Then the node at index 1 ("A") will have its `next` field adjusted to move around the node to be removed ("F"). The modified list will look like this:



Since there is no longer a reference to the node with "F", the memory for that node will be reclaimed by Java's garbage collector. When the method is finished, the local variable `ref` also disappears and the list will look like this:



The `removeElementAt` method is left as a programming exercise.

Deleting an element from a Linked List: `remove`

When deleting an element from a linked list, the code in this particular class must recognize these two cases:

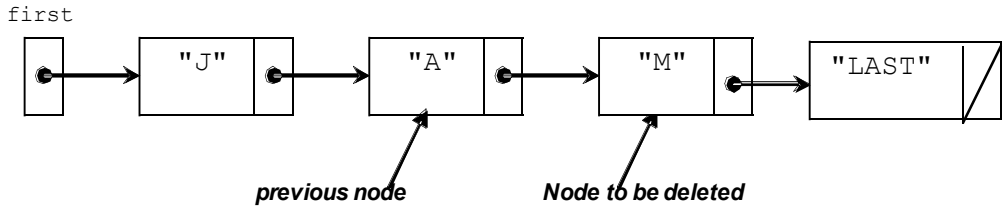
1. Deleting the first element from the list (a special case again)
2. Deleting an interior node from the list (including the last node)

When deleting the first node from a linked list, care must be taken to ensure that the rest of the list is not destroyed. The adjustment to `first` is again necessary so that all the other methods work (and the object is not in a corrupt state). This can be accomplished by shifting the reference value of `first` to the second element of the list (or to `null` if there is only one element). One assignment will do this:

```
first = first.next;
```

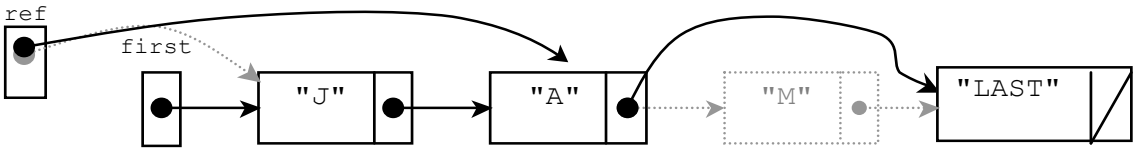
Now consider removing a specific element that may or may not be stored in an interior node. As with `removeElementAt`, the code will look to place a reference to the node just *before* the node to be removed. So to remove `"M"`, the link to the node before M is needed. This is the node with `"A"`.

```
list.remove("M");
```



At this point, the `next` field in the node with "A" can be "sent around" the node to be removed ("M"). Assuming the `Node` variable named ref is storing the reference to the node before the node to be deleted, the following assignment effectively removes a node and its element "M" from the structure:

```
ref.next = ref.next.next;
```



Deleting a specific internal node, in this case "M"

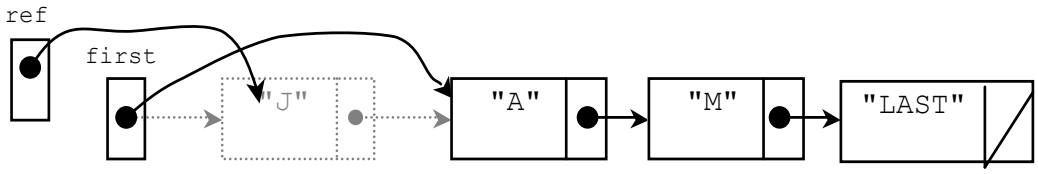
This results in the removal of an element from the interior of the linked structure. The memory used to store the node with "M" will be reclaimed by Java's garbage collector.

The trick to solving this problem is comparing the data that is one node ahead. Then you must make a reference to the node before the found element (assuming it exists in the list). The following code does just that. It removes the first occurrence of the `objectToRemove` found in the linked list. It uses the class's `equals` method to make sure that the element located in the node equals the state of the object that the message intended to remove. First, a check is made for an empty list.

```
/**
  * Remove element if found using the equals method for type E.
  * @param The object to remove from this list if found
  */
  public boolean remove(E element) {
    boolean result = true;

    // Don't attempt to remove an element from an empty list
    if (this.isEmpty())
      result = false;
```

The following code takes care of the special case of removing the first element when the list is not empty:

```
    else {
      // If not empty, begin to search for an element that equals obj
      // Special case: Check for removing first element
      if (first.data.equals(element))
        first = first.next;
```



Checking for these special cases has an added advantage. The algorithm can now assume that there is at least one element in the list. It can safely proceed to look one node ahead for the element to remove. A `while` loop traverses the linked structure while comparing `objectToRemove` to the `data` in the node one element ahead. This traversal will terminate when either of these two conditions occur:

1. The end of the list is reached.
2. An item in the next node `equals` the element to be removed.

The algorithm assumes that the element to be removed is in index 1 through `size()-1` (or it's not there at all). This allows the `Node` variable named `ref` to "peek ahead" one node. Instead of comparing `objectToRemove` to `ref.data`, `objectToRemove` is compared to `ref.next.data`.

```
  else {
    // Search through the rest of the list
    Node ref = first;
    // Look ahead one node
    while ((ref.next != null) && !(element.equals(ref.next.data)))
      ref = ref.next;
```

This `while` loop handles both loop termination conditions. The loop terminates when `ref`'s `next` field is `null` (the first expression in the loop test). The loop will also terminate when the next element (`ref.next.data`) in the list `equals(objectToRemove)`, the element to be removed. Writing the test for `null` before the equals message avoids `null` pointer exceptions. Java's guaranteed short circuit boolean evaluation will not let the expression after && execute when the first subexpression (`ref.next != null`) is false.

Self-Check

15-8 What can happen if the subexpressions in the loop test above are reversed?

```
while (!(objectToRemove.equals(ref.next.data))
          && (ref.next != null))
```

At the end of this loop, `ref` would be pointing to one of two places:

1. the node just before the node to be removed, or
2. the last element in the list.

In the latter case, no element "equaled" `objectToRemove`. Because there are two ways to terminate the loop, a test is made to see if the removal element was indeed in the list. The link adjustment to remove a node executes only if the loop terminated before the end of the list was reached. The following code modifies the list only if `objectToRemove` was found.

```
    // Remove node if found (ref.next != null).  However, if
    // ref.next is null, the search stopped at end of list.
    if (ref.next == null)
      return false;  // Got to the end without finding element
    else {
      ref.next = ref.next.next;
      return true;
    }
  } // end remove(E element)
```

Self-Check

15-9 In the space provided, write the expected value that would make the assertions pass:

```
OurLinkedList<String> list = new OurLinkedList<String>();
list.addLast("A");
list.insertElementAt(0, "B");
list.addFirst("C");
assertEquals(_______________, list.toString());  // a.
list.remove("B");
assertEquals(_______________, list.toString());  // b.
list.remove("A");
assertEquals(_______________, list.toString());  // c.
list.remove("Not Here");
assertEquals(_______________, list.toString());  // d.
list.remove("C");
assertEquals(_______________, list.toString());  // e.
```

15-10 What must you take into consideration when executing the following code?

```
if (current.data.equals("CS 127B"))
  current.next.data = "CS 335";
```

15.3 When to use Linked Structures

The one advantage of a linked implementation over an array implementation may be constrained to the growth and shrink factor when adding elements. With an array representation, growing an array during `add` and shrinking an array during `remove` requires an additional temporary array of contiguous memory be allocated. Once all elements are copied, the memory for the temporary array can be garbage collected. However, for that moment, the system has to find a large

contiguous block of memory. In a worst case scenario, this could potentially cause an `OutOfMemoryException`.

When adding to a linked list, the system allocates the needed object and reference plus an additional 4 bytes overhead for the `next` reference value. This may work on some systems better than an array implementation, but it is difficult to predict which is better and when.

The linked list implementation also may be more time efficient during inserts and removes. With an array, removing the first element required n assignments. Removing from a linked list requires only one assignment. Removing an internal node may also run a bit faster for a linked implementation, since the worst case rarely occurs. With an array, the worst case always occurs—n operations are needed no matter which element is being removed. With a linked list, it may be more like n/2 operations.

Adding another external reference to refer to the last element in a linked list would make the addLast method run O(1), which is as efficient as an array data structure. A linked list can also be made to have links to the node before it to allow two-way traversals and faster removes — a doubly linked list. This structure could be useful in some circumstances.

A good place to use linked structures will be shown in the implementation of the stack and queue data structures in later chapters. In both collections, access is limited to one or both ends of the collection. Both grow and shrink frequently, so the memory and time overhead of shifting elements are avoided (however, an array can be used as efficiently with a few tricks).

Computer memory is another thing to consider when deciding which implementation of a list to use. If an array needs to be "grown" during an `add` operation, for a brief time there is a need for twice as many reference values. Memory is needed to store the references in the original array. An extra temporary array is also needed. For example, if the array to be grown has an original capacity of 50,000 elements, there will be a need for an additional 200,000 bytes of memory until the references in the original array are copied to the temporary array. Using a linked list does not require as much memory to grow. The linked list needs as many references as the array does for each element, however at grow time the linked list can be more efficient in terms of memory (and time). The linked list does not need extra reference values when it grows.

Consider a list of 10,000 elements. A linked structure implementation needs an extra reference value (`next`) for every element. That is overhead of 40,000 bytes of memory with the linked version. An array-based implementation that stores 10,000 elements with a capacity of 10,000 uses the same amount of memory. Imagine the array has 20 unused array locations — there would be only 80 wasted bytes. However, as already mentioned, the array requires double the amount of overhead when growing itself. Linked lists provide the background for another data structure called the binary tree structure in a later chapter.

When not to use Linked Structures

If you want quick access to your data, a linked list will not be that helpful when the size of the collection is big. This is because accessing elements in a linked list has to be done sequentially. To maintain a fixed list that has to be queried a lot, the algorithm needs to traverse the list each time in order to get to the information. So if a lot of `set` and `gets` are done, the array version tends to be faster. The access is O(1) rather than O(n). Also, if you have information in an array that is sorted, you can use the more efficient binary search algorithm to locate an element.

A rather specific time to avoid linked structures (or any dynamic memory allocations) is when building software for control systems on aircraft. The United States Federal Aviation Association (FAA) does not allow it because it's not safe to have a airplane system run out of memory in flight. The code must work with fixed arrays. All airline control code is carefully reviewed to ensure that allocating memory at runtime is not present. With Java, this would mean there could never be any existence of `new`.

One reason to use the linked version of a list over an array-based list is when the collection is very large and there are frequent add and removal messages that trigger the grow array and shrink array loops. However, this could be adjusted by increasing the `GROW_SHRINK_INCREMENT` from 20 to some higher number. Here is a comparison of the runtimes for the two collection classes implemented over this and the previous chapter.

	`OurArrayList`	`OurLinkedList`
get and set	O(1)	O(n)
remove removeElementAt	O(n)	O(n)
find[4]	O(n)	O(n)
add(int index, Object el)	O(n)	O(n)
size[5]	O(1)	O(n)
addFirst	O(n)	O(1)
addLast[6]	O(1)	O(n)

One advantage of arrays is the `get` and `set` operations of `OurArrayList` are an order of magnitude better than the linked version. So why study singly linked structures?

1. The linked structure is a more appropriate implementation mechanism for the stacks and queues of the next chapter.
2. Singly linked lists will help you to understand how to implement other powerful and efficient linked structures (trees and skip lists, for example).

When a collection is very large, you shouldn't use either of the collection classes shown in this chapter, or even Java's `ArrayList` or `LinkedList` classes in the `java.util` package. There are other data structures such as hash tables, heaps, and trees for large collections. These will be discussed later in this book. In the meantime, the implementations of a list interface provided insights into the inner workings of collections classes and two storage structures. You have looked at collections from both sides now.

Self-Check

15-11 Suppose you needed to organize a collection of student information for your school's administration. There are approximately 8,000 students in the university with no significant growth expected in the coming years. You expect several hundred lookups on the collection everyday. You have only two data structures to store the data, and array and a linked structure. Which would you use? Explain.

---

[4] `find` could be improved to O(log n) if the data structure is changed to an ordered and sorted list.
[5] `size` could be improved to O(1) if the `SimpleLinkedList` maintained a separate instance variable for the number of element in the list (add 1during inserts subtract 1 during successful removes)
[6] `addLast` with `SimpleLinkedList` could be improved by maintaining an external reference to the last element in the list.

Answers to Self-Checks

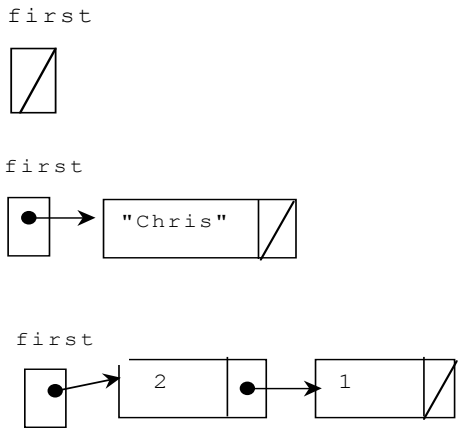
15-1 first.data.equals("Bob")

15-2 first.next.data ("Chris");

15-3 first.next.next.next.data.equals("Zorro");

15-4 first.next.next.next refers to a Node with a reference to "Zorro" and null in its next field.

15-5 drawing of memory



15-6 c would fail ("JO" should be "Jo")

15-7 which would throw an exception

a IndexOutOfBoundsException

c the largest valid index is currently 4

d Okay since the largest index can be the size, which is 4 in this case

15-8 if switched, ref would move one Node too far and cause a NullPointerException

15-9 assertions - listed in correct order

```
OurLinkedList<String> list = new OurLinkedList<String>();
list.addLast("A");
list.insertElementAt(0, "B");
list.addFirst("C");
assertEquals(__"[C, B, A]"__, list.toString()); // a.
list.remove("B");
assertEquals(___"[C, A]"__, list.toString());  // b.
list.remove("A");
assertEquals(____"[C]"_____, list.toString()); // c.
list.remove("Not Here");
assertEquals(____"[C]"_____, list.toString()); // d.
list.remove("C");
assertEquals(_____"[]"_____, list.toString()); // e.
```

15-10 Whether or not a node actually exists at current.next. It could be null.

15-11 An array so the more efficient binary search could be used rather than the sequential search necessary with a linked structure.

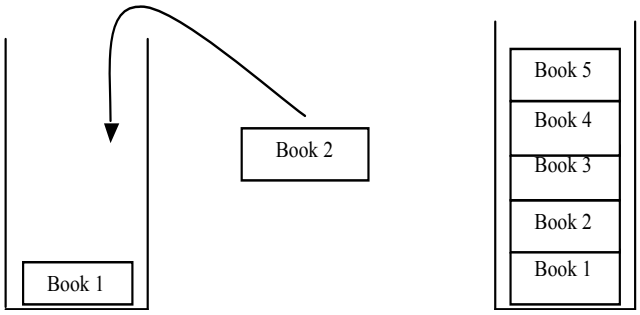
Chapter 16

Stacks and Queues

16.1 Stacks

The stack abstract data type allows access to only one element—the one most recently added. This location is referred to as the top of the stack.

Consider how a stack of books might be placed into and removed from a cardboard box, assuming you can only move one book at a time. The most readily available book is at the top of the stack. For example, if you add two books—Book 1 and then Book 2—into the box, the most recently added book (Book 2) will be at the top of the stack. If you continue to stack books on top of one another until there are five, Book 5 will be at the top of the stack. To get to the least recently added book (Book 1), you first remove the topmost four books (Book 5, Book 4, Book 3, Book 2) — one at a time. Then the top of the stack would be Book 1 again.



Stack elements are added and removed in a last in first out (LIFO) manner. The most recent element added to the collection will be the first element to be removed from the collection. Sometimes, the only data that is readily needed is the most recently accessed one. The other elements, if needed later, will be in the reverse order of when they were pushed. Many real world examples of a stack exist. In a physical sense, there are stacks of books, stacks of cafeteria trays, and stacks of paper in a printer's paper tray. The sheet of paper on the top is the one that will get used next by the printer.

For example, a stack maintains the order of method calls in a program. If `main` calls `function1`, that method calls `function2`, which in turn calls `function3`. Where does the program control go to when `function3` is finished? After `function3` completes, it is removed from the stack as the most recently added method. Control then returns to the method that is at the new top of the stack — `function2`.

Here is a view of the stack of function calls shown in a thread named `main`. This environment (Eclipse) shows the first method (`main`) at the bottom of the stack. `main` will also be the last method popped as the program finishes — the *first* method called is the *last* one to execute. At all other times, the method on the top of the stack is executing. When a method finishes, it can be removed and the method that called it will be the next one to be removed form the stack of method calls.

```
3  public class StackedMethods {
4
5    void methodThree() {
6      out.println("Four method calls are on the stack");
7    }
8
9    void methodTwo() {
10     methodThree();
11     out.println("Two about to end");
12   }
13
14   void methodOne() {
15     methodTwo();
16     out.println("One about to end");
17   }
18
19   public static void main(String[] args) {
20     StackedMethods sm = new StackedMethods();
21     sm.methodOne();
22     out.println("main about to end");
23   }
24 }
```

Problems | Javadoc | Declaration | Console | Debug | Search

- StackedMethods [Java Application]
  - StackedMethods at localhost:1037
    - Thread [main] (Suspended (breakpoint at line 6 in StackedMethods))
      - StackedMethods.methodThree() line: 6
      - StackedMethods.methodTwo() line: 10
      - StackedMethods.methodOne() line: 15
      - StackedMethods.main(String[]) line: 21

Writable | Smart Insert | 6 : 1

The program output indicates the last in first out (or first in last out) nature of stacks:

```
Four method calls are on the stack
Two about to end
One about to end
main about to end
```

Another computer-based example of a stack occurs when a compiler checks the syntax of a program. For example, some compilers first check to make sure that `[ ]`, `{ }`, and ( ) are balanced properly. Thus, in a Java `class`, the final `}` should match the opening `{`. Some compilers do this type of symbol balance checking first (before other syntax is checked) because incorrect matching could otherwise lead to numerous error messages that are not really errors. A stack is a natural data structure that allows the compiler to match up such opening and closing symbols (an algorithm will be discussed in detail later).

A Stack Interface to capture the ADT

Here are the operations usually associated with a stack. (As shown later, others may exist):

- `push` place a new element at the "top" of the stack
- `pop` remove the top element and return a reference to the top element
- `isEmpty` return `true` if there are no elements on the stack
- `peek` return a reference to the element at the top of the stack

Programmers will sometimes add operations and/or use different names. For example, in the past, Sun programmers working on Java collection classes have used the name `empty` rather than `isEmpty`. Also, some programmers write their stack class with a `pop` method that does not return a reference to the element at the top of the stack. Our `pop` method will modify and access the state of stack during the same message.

Again, a Java interface helps specify Stack as an abstract data type. For the discussion of how a stack behaves, consider that `LinkedStack` (a collection class) implements the `OurStack` interface, which is an ADT specification in Java.

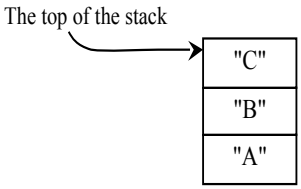
```
import java.util.EmptyStackException;

public interface OurStack<E> {
/** Check if the stack is empty to help avoid popping an empty stack.
  * @returns true if there are zero elements in this stack.
  */
  public boolean isEmpty();

/** Put element on "top" of this Stack object.
  * @param The new element to be placed at the top of this stack.
  */
  public void push(E element);

/** Return reference to the element at the top of this stack.
  * @returns A reference to the top element.
  * @throws EmptyStackException if the stack is empty.
  */
  public E peek() throws EmptyStackException;

/** Remove element at top of stack and return a reference to it.
  * @returns A reference to the most recently pushed element.
  * @throws EmptyStackException if the stack is empty.
  */
  public E pop() throws EmptyStackException;
}
```

You might need a stack of integers, or a stack of string values, or a stack of some new class of `Token` objects (pieces of source code). One solution would be to write and test a different stack class for each new class of object or primitive value that you want to store. This is a good reason for developing an alternate solution— a generic stack.

The interface to be implemented specifies the operations for a stack class. It represents the *abstract* specification. There is no particular data storage mentioned and there is no code in the methods. The type parameter `<E>` and return types `E` indicate that the objects of the implementing class will store any type of element. For example, `push` takes an `E` parameter while `peek` and `pop` return an `E` reference.

The following code demonstrates the behavior of the stack class assuming it is implemented by a class named `LinkedStack`.

```
// Construct an empty stack that can store any type of element
OurStack stackOfStrings<String> = new LinkedStack<String>();
// Add three string values to the stack
stackOfStrings.push("A");
stackOfStrings.push("B");
stackOfStrings.push("C");
// Show each element before removal in a LIFO order
while (! stackOfStrings.isEmpty()) {
  // Print the value at the top as it is removed
  System.out.print(stackOfStrings.pop() + " ");
}
```



Output:

```
C B A
```

Self-Check

16-1 Write the output generated by the following code:

```
OurStack<String> aStack = new LinkedStack<String> ();
aStack.push("x");
aStack.push("y");
aStack.push("z");
while (! aStack.isEmpty()) {
  out.println(aStack.pop());
}
```

16-2 Write the output generated by the following code:

```
OurStack<Character> opStack = new OurLinkedStack<Character>();
System.out.println(opStack.isEmpty());
opStack.push('>');
opStack.push('+');
opStack.push('<');
out.print(opStack.peek());
out.print(opStack.peek());  // careful
out.print(opStack.peek());
```

16-3 Write the output generated by the following code:

```
OurStack<Integer> aStack = new OurLinkedStack<Integer>();
aStack.push(3);
aStack.push(2);
aStack.push(1);
System.out.println(aStack.isEmpty());
System.out.println(aStack.peek());
aStack.pop();
System.out.println(aStack.peek());
aStack.pop();
System.out.println(aStack.peek());
aStack.pop();
System.out.println(aStack.isEmpty());
```

16.2 Stack Application: Balanced Symbols

Some compilers perform symbol balance checking before checking for other syntax errors. For example, consider the following code and the compile time error message generated by a particular Java compiler (your compiler may vary).

```
public class BalancingErrors
  public static void main(String[] args) {
    int x = p;
    int y = 4;
    in z = x + y;
    System.out.println("Value of z = " + z);
  }
}
```

```
BalancingErrors.java:1: '{' expected
public class BalancingErrors
                            ^
```

Notice that the compiler did not report other errors, one of which is on line 3. There should have been an error message indicating `p` is an unknown symbol. Another compile time error is on line 5 where `z` is incorrectly declared as an `in` not `int`. If you fix the first error by adding the left curly brace on a new line 1 you will see these other two errors.

```
public class BalancingErrors  {  // <- add an opening curly brace
  public static void main(String[] args) {
    int x = p;
    int y = 4;
    in z = x + y;
    System.out.println("Value of z = " + z);
  }
}
```

```
BalancingErrors.java:3: cannot resolve symbol
symbol  : variable p
location: class BalancingErrors
    int x = p;
            ^
BalancingErrors.java:5: cannot resolve symbol
symbol  : class in
location: class BalancingErrors
    in z = x + y;
    ^
2 errors
```

This behavior could be due to a compiler that first checks for balanced { and } symbols before looking for other syntax errors.

Now consider how a compiler might use a stack to check for balanced symbols such as ( ) , `{ }`, and `[ ]`. As it reads the Java source code, it will only consider opening symbols: `( { [`, and closing symbols:) `} ]`. If an opening symbol is found in the input file, it is pushed onto a stack. When a closing symbol is read, it is compared to the opener on the top of the stack. If the symbols match, the stack gets popped. If they do not match, the compiler reports an error to the programmer. Now imagine processing these tokens, which represent only the openers and closers in a short Java program: `{{([])}}`. As the first four symbols are read — all openers — they get pushed onto the stack.

Java source code starts as: `{{([])}}`

```
[
(   push the first four opening symbols with [ at the top. Still need to read ])}}
{
{
```

The next symbol read is a closer: `"]"`. The `"["` would be popped from the top of the stack and compared to `"]"`. Since the closer matches the opening symbol, no error would be reported. The stack would now look like this with no error reported:

```
(
{    pop [ which matches ]. There is no error. Still need to read ) }}
{
```

The closing parenthesis `")"` is read next. The stack gets popped again. Since the symbol at the top of the stack `"("` matches the closer `")"`, no error needs to be reported. The stack would now have the two opening curly braces.

```
{                pop ( which matches). There is no error. Still need to read }}
{
```

The two remaining closing curly braces would cause the two matching openers to be popped with no errors. It is the last-in-first-out nature of stacks that allows the first pushed opener `"{"` to be associated with the last closing symbol `"}"` that is read.

Now consider Java source code with only the symbols `(]`. The opener `"("` is pushed. But when the closer `"]"` is encountered, the popped symbol `"("` does not match `"]"` and an error could be reported. Here are some other times when the use of a stack could be used to help detect unbalanced symbols:

8. If a closer is found and the stack is empty. For example, when the symbols are `{}}`. The opening `{` is pushed and the closer `"}"` is found to be correct. However when the second `}` is encountered, the stack is empty. There is an error when `}` is discovered to be an extra closer (or perhaps `{` is missing).
9. If all source code is read and the stack is not empty, an error should be reported. This would happen with Java source code of `{{([])}`. In this case, there is a missing right curly brace. Most symbols are processed without error. At the end, the stack *should* be empty. Since the stack is *not* empty, an error should be reported to indicate a missing closer.

This algorithm summarizes the previous actions.

```
1. Make an empty stack named s
2. Read symbols until end of file
     if it's an opening symbol, push it
     if it is a closing symbol && s.empty
        report error
    otherwise
        pop the stack
        if symbol is not a closer for pop's value, report error
 3. At end of file, if !s.empty, report error
```

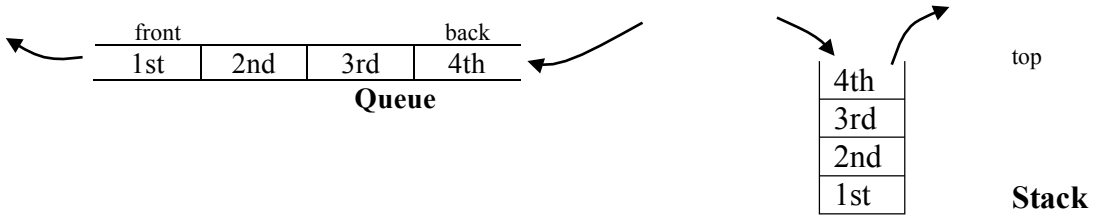
Self-Check

16-4 Write the errors generated when the algorithm above processes the following input file:

```
public class Test2 {
   public static void main(String[] args) {
     System.out.println();
  )) {
```

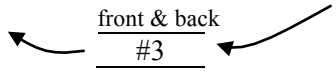
16.3 FIFO Queues

A first-in, first-out (FIFO) **queue** — pronounced "Q" — models a waiting line. Whereas stacks add and remove elements at one location — the `top` — `queues` add and remove elements at different locations. New elements are added at the back of the queue. Elements are removed from the front of the queue.

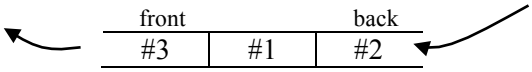


Whereas stacks mimic LIFO behavior, queues mimic a first in first out (FIFO) behavior. So, for example, the queue data structure models a waiting line such as people waiting for a ride at an amusement park. The person at the front of the line will be the first person to ride. The most recently added person must wait for all the people in front of them to get on the ride. With a FIFO queue, the person waiting longest in line is served before all the others who have waited less time[7].

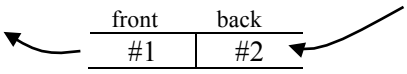
Another example of queue behavior can be found when several documents need to be printed at a shared printer. Consider three students, on the same network, trying to print one document each. Who gets their document printed first? If a FIFO queue is being used to store incoming print requests, the student whose request reached the print queue first will get printed ahead of the others. Now assume that the printer is busy and the print queue gets a print request from student #3 while a ducument is printing. The print queue would look something like this:



In this case the queue's front element is also at the back end of the queue. The queue contains one element. Now add another request from student #1, followed by another request from student #2 for printing, and the print queue would look like this:



Student #1 and student #2 requests were added to the back of queue. The print requests are stored in the order in which they arrived. As the printer prints documents, the document will be removed from the front. Once the printer has printed the current document, the document for student #3 will then be removed. Then the state of the queue will now look like this:



A Queue Interface — Specifying the methods

[7] Note: A *priority queue* has different behavior where elements with a higher priority would be removed first. For example, the emergency room patient with the most need is attended to next, not the patient who has been there the longest.

There is no universally agreed upon set of operations; however the following is a reasonable set of operations for a FIFO Queue ADT.

- isEmpty Return true only when there are zero elements in the queue
- add Add an element at the back of the queue
- peek Return a reference to the element at the front of the queue
- remove Return a reference to the element at the front and remove the element

This leads to the following interface for a queue that can store any class of object.

```
public interface OurQueue<E> {

  /**
   * Find out if the queue is empty.
   * @returns true if there are zero elements in this queue.
   */
  public boolean isEmpty();

  /**
   * Add element to the "end" of this queue.
   * @param newEl element to be placed at the end of this queue.
   */
  public void add(E newEl);

  /**
   * Return a reference to the element at the front of this queue.
   * @returns A reference to the element at the front.
   * @throws NoSuchElementException if this queue is empty.
   */
  public E peek();

  /**
   * Return a reference to front element and remove it.
   * @returns A reference to the element at the front.
   * @throws NoSuchElementException if this queue is empty.
   */
  public E remove();
}
```

The following code demonstrates the behavior of the methods assuming OurLinkedQueue implements interface OurQueue:

```
OurQueue<Integer> q = new OurLinkedQueue<Integer>();
q.add(6);
q.add(2);
q.add(4);
while (!q.isEmpty()) {
  System.out.println(q.peek());
  q.remove();
}
```

Output

```
6 2 4
```

Self-Check

16-5 Write the output generated by the following code.

```
OurQueue<String> stringQueue = new OurLinkedQueue<String>();
stringQueue.add("J");
stringQueue.add("a");
stringQueue.add("v");
stringQueue.add("a");
while (!stringQueue.isEmpty()) {
  System.out.print(stringQueue.remove());
}
```

16-6 Write the output generated by the following code until you understand what is going on.

```
OurQueue<String> stringQueue = new OurLinkedQueue<String>();
stringQueue.add("first");
stringQueue.add("second");
while (! stringQueue.isEmpty()) {
  System.out.println(stringQueue.peek());
}
```

16-7 Write code that displays a message to indicate if each integer in a queue named `intQueue` is even or odd. The queue must remain intact after you are done. The queue is initialized with random integers in the range of 0 through 99.

```
OurQueue<Integer> intQueue = new OurLinkedQueue<Integer>();
Random generator = new Random();
for(int j = 1; j <= 7; j++) {
  intQueue.add(generator.nextInt(100));
}
// Your solution goes here
```

Sample Output (output varies since random integers are added)

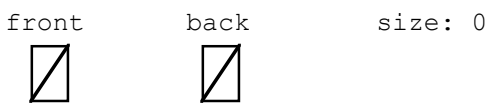
```
28 is even
72 is even
4 is even
37 is odd
94 is even
98 is even
33 is odd
```

16.4 Queue with a Linked Structure

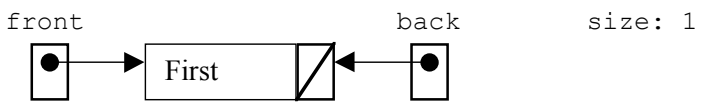
We will implement interface OurQueue with a class that uses a singly linked structure. There are several reasons to choose a linked structure over an array:

- It is easier to implement. (A programming project explains the trickier array-based implementation).
- The Big-O runtime of all algorithms is as efficient as if an array were used to store the elements. All algorithms can be O(1).
- An array-based queue would have `add` and `remove` methods, which will occasionally run O(n) rather than O(1). This occurs whenever the array capacity needs to be increased or decreased.
- It provides another good example of implementing a data structure using the linked structure introduced in the previous chapter.

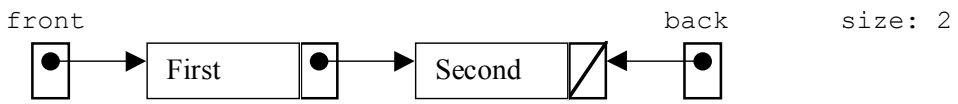
Elements are removed from the "front" of a queue. New elements are added at the back of the queue. Both "ends" of the queue are frequently accessed. Therefore, this implementation of `OurQueue` will use two external references. Only one external reference to the front is required. However, this would make for O(n) behavior during `add` messages, since a loop would need to sequence through all elements before reaching the end. With only a reference to the front, all elements must be visited to find the end of the list before one could be added. Therefore, an external reference named `back` will be maintained in addition to front. This will allow `add` to be O(1). An empty OurLinkedQueue will look like this:



After `q.add("First")`, a queue of size 1 will look like this:



After `q.add("Second")`, the queue of size 2 will look like this:



This test method shows the changing state of a queue that follows the above pictures of memory.

```
@Test
public void testAddAndPeek() {
  OurQueue<String> q = new OurLinkedQueue<String>();
  assertTrue(q.isEmpty()); // front == null
  q.add("first");
  assertEquals("first", q.peek());  // front.data is "first"
  assertFalse(q.isEmpty());

  q.add("second");  // Change back, not front
  // Front element should still be the same
  assertEquals("first", q.peek());
}
```

The first element is accessible as `front.data`. A new element is added by storing a reference to the new node into `back.next` and adjusting `back` to reference the new node at the end.

Here is the beginning of class OurLinkedQueue that once again uses a private inner `Node` class to store the data along with a link to the next element in the collection. There are two instance variables to maintain both ends of the queue.

```
public class OurLinkedQueue<E> implements OurQueue<E> {

  private class Node {
    private E data;
    private Node next;

    public Node() {
      data = null;
      next = null;
    }

    public Node(E elementReference, Node nextReference) {
      data = elementReference;
      next = nextReference;
    }
  } // end class Node

  // External references to maintain both ends of a Queue
  private Node front;
  private Node back;

  /**
   * Construct an empty queue (no elements) of size 0.
   */
  public OurLinkedQueue() {
    front = null;
    back = null;
  }

 /** Find out if the queue is empty.
  * @returns true if there are zero elements in this queue.
  */
  public boolean isEmpty()  {
    return front == null;
  }

  // More methods to be added . . .
}
```

This implementation recognizes an empty queue when `front` is `null`.

add

The `add` operation will first check for the special case of adding to an empty queue. The code to add to a non-empty queue is slightly different. If the queue is empty, the external references front and back are both `null`.

In the case of an empty queue, the single element added will be at front of the queue and also at

the back of the queue. So, after building the new node, front and back should both refer to the same node. Here is a before and after picture made possible with the code shown.

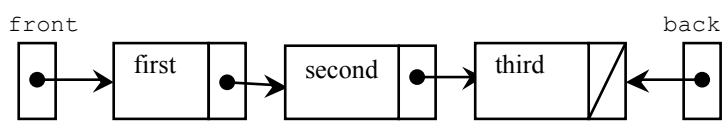
```
// Build a node to be added at the end. A queue can
// grow as long as the computer has enough memory.
// With a linked structure, resizing is not necessary.
if (this.isEmpty()) {
   front = new Node(element, null);
   back = front;
 }
```

When an add messages is sent to a queue that is not empty, the last node in the queue must be made to refer to the node with the new element. Although `front` must remain the same during `add` messages, `back` must be changed to refer the new element at the end.

```
else {
  back.next = new Node(element);
  back = back.next;
}
```

There are several viable variations of how algorithms could be implemented when a linked structure is used to store the collection of elements. The linked structure used here always maintains two external references for the `front` and `back` of the linked structure. This was done so `add` is O(1) rather than O(n). In summary, the following code will generate the linked structure shown below.

```
OurQueue<String> q = new OurLinkedQueue<String>();
q.add("first");
q.add("second");
q.add("third");
```



Self-Check

16-8 Draw a picture of what the memory would look like after this code has executed

```
OurQueue<Double> q1 = new OurLinkedQueue<Double>();
q1.add(5.6));
q1.add(7.8));
```

16-9 Implement a `toString` method for `OurLinkedQueue` so this assertion would pass after the code in the previous self-check question:

```
assertEquals("[a, b]", q2.toString());
```

peek

The `peek` method throws a `NoSuchElementException` if the queue is empty. Otherwise, `peek` returns a reference to the element stored in front.data.

```
/**
 * Return a reference to the element at the front of this queue.
 * @returns A reference to the element at the front.
 * @throws NoSuchElementException if this queue is empty.
 */
public E peek() {
   if (this.isEmpty())
     throw new java.util.NoSuchElementException();
   else
     return front.data;
 }
```

The next two test methods verify that `peek` returns the expected value and that it does not modify the queue.

```
@Test
public void testPeek() {
  OurQueue<String> q = new OurLinkedQueue<String>();
  q.add(new String("first"));
  assertEquals("first", q.peek());
  assertEquals("first", q.peek());

  OurQueue<Double> numbers = new OurLinkedQueue<Double>();
  numbers.add(1.2);
  assertEquals(1.2, numbers.peek(), 1e-14);
  assertEquals(1.2, numbers.peek(), 1e-14);
}

@Test
public void testIsEmptyAfterPeek() {
  OurQueue<String> q = new OurLinkedQueue<String>();
  q.add("first");
  assertFalse(q.isEmpty());
  assertEquals("first", q.peek());
}
```

An attempt to peek at the element at the front of an empty queue results in a `java.util.NoSuchElementException`, as verified by this test:

```
@Test(expected = NoSuchElementException.class)
public void testPeekOnEmptyList() {
  OurQueue<String> q = new OurLinkedQueue<String>();
  q.peek();
}
```

remove

The `remove` method will throw an exception if the queue is empty. Otherwise, `remove` returns a reference to the object at the front of the queue (the same element as `peek()` would). The `remove` method also removes the front element from the collection.

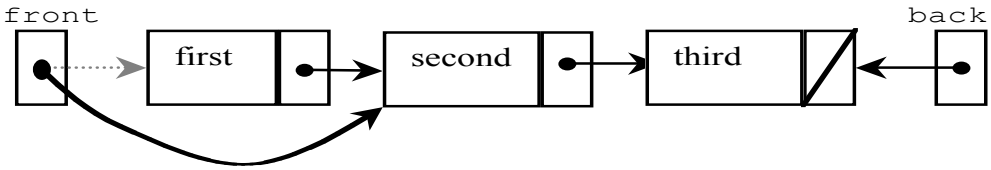
```
@Test
 public void testRemove() {
   OurQueue<String> q = new OurLinkedQueue<String>();
   q.add("c");
   q.add("a");
   q.add("b");
   assertEquals("c", q.remove());
   assertEquals("a", q.remove());
   assertEquals("b", q.remove());
 }

 @Test(expected = NoSuchElementException.class)
 public void testRemoveThrowsAnException() {
   OurQueue<Integer> q = new OurLinkedQueue<Integer>();
   q.remove();
 }
```

Before the front node element is removed, a reference to the front element must be stored so it can be returned after removing it.

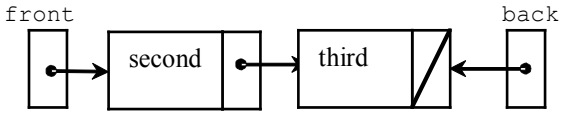
```
E frontElement = front.data;
```

`front`'s next field can be sent around the first element to eliminate it from the linked structure.

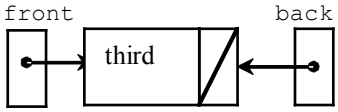
```
front = front.next;
```



Now the method can return a reference to `firstElement`. The linked structure would now look like this.



Another remove makes the list look like this.



Another remove message will return "third". The `remove` method should set front to null so isEmpty() will still work. This will leave the linked structure like this with back referring to a node that is no longer considered to be part of the queue. In this case, back will also be set to null.

Self-Check

16-10 Complete method `remove` so it return a reference to the element at the front of this queue while removing the front element. If the queue is empty, `throw new NoSuchElementException()`.

```
public E remove() {
```

Answers to Self-Checks

16-1
```
z
y
x
```

16-2
```
true
<<<
```

16-3
```
false
1
2
3
true
```

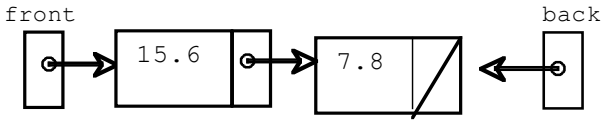
16-4
```
Check symbols in Test2.java
Abc.java:2 expecting ]
Abc.java:4 expecting }
Abc.java:4 expecting }
missing }
4 errors
```

16-5
```
Java
```

16-6
```
first
first
first
  ...
first until someone terminates the program or the power goes out
```

16-7
```
int size = intQueue.size();
for (int j = 1; j <= size; j++) {
  int nextInt = intQueue.peek();
  if (nextInt % 2 != 0)
     System.out.println(nextInt + " is odd");
  else
     System.out.println(nextInt + " is even");
  intQueue.remove();
  intQueue.add(nextInt);
}
```

16-8



16-9
```
public String toString() {
   String result = "[";

   // Concatenate all but the last one (if size > 0)
   Node ref = front;
   while (ref != back) {
     result += ref.data + ", ";
     ref = ref.next;
   }

   // Last element does not have ", " after it
   if (ref != null)
     result += ref.data;

   result += "]";

   return result;
}
```

16-10
```
public E remove() throws NoSuchElementException {
   if (this.isEmpty())
     throw new NoSuchElementException();

   E frontElement = front.data;
   front = front.next;

   if (front == null)
     front = back = null;

   return frontElement;
}
```

Chapter 17

Recursion

Goals

- Trace recursive algorithms
- Implement recursive algorithms

17.1 Simple Recursion

One day, an instructor was having difficulties with a classroom's multimedia equipment. The bell rang, and still there was no way to project the lecture notes for the day. To keep her students occupied while waiting for the AV folks, she asked one of her favorite students, Kelly, to take attendance. Since the topic of the day was recursion, the instructor proposed a recursive solution: Instead of counting each student in the class, Kelly could count the number of students in her row and remember that count. The instructor asked Kelly to ask another student in the row behind her to do the same thing—count the number of students in their row, remember the count, and ask the same question of the next row.

By the time the question reached the last row of seats in the room, there was one person in each row who knew the number of students in that particular row. Andy, who had just counted eight students in the last row, asked his instructor what to do since there were no more rows of seats behind him. The teacher responded that all he had to do was return the number of students in his row to the person who asked him the question moments ago. So, Andy gave his answer of eight to the person in the row in front of him.

The student in the second to last row added the number of people in her row (12) to the number of students in the row behind her, which Andy had just told her (8). She returned the sum of 20 to the person in the row in front of her.

At that point the AV folks arrived and discovered a bent pin in a video jack. As they were fixing this, the students continued to return the number of students in their row plus the number of students behind them, until Kelly was informed that there were 50 students in all the rows behind her. At that point, the lecture notes, entitled "Recursion", were visible on the screen. Kelly told her teacher that there were 50 students behind her, plus 12 students in her first row, for a total of 62 students present.

The teacher adapted her lecture. She began by writing the algorithm for the head count problem. Every row got this same algorithm.

```
if you have rows behind you
    return the number of students in your row plus the number behind you
otherwise
    return the number of students in your row
```

Andy asked why Kelly couldn't have just counted the students one by one. The teacher responded, "That would be an *iterative* solution. Instead, you just solved a problem using a *recursive* solution. This is precisely how I intend to introduce recursion to you, by comparing recursive solutions to problems that could also be solved with iteration. I will suggest to you that some problems are better handled by a recursive solution."

Recursive solutions have a final situation when nothing more needs to be done—this is the base case—and situations when the same thing needs to be done again while bringing the problem closer to a base case. Recursion involves partitioning problems into simpler subproblems. It requires that each subproblem be identical in structure to the original problem.

Before looking at some recursive Java methods, consider a few more examples of recursive solutions and definitions. Recursive definitions define something by using that something as part of the definition.

Recursion Example 1

Look up a word in a dictionary:

find the word in the dictionary
if there is a word in the definition that you do not understand
look up that word in the dictionary

Example: Look up the term **object**

Look up **object**, which is defined as "an instance of a **class**."

What is a class? *Look up* **class** to find "a collection of **method**s and data."

What is a method? *Look up* **method** to find "a **method heading** followed by a collection of programming statements."

Example: Look up the term **method heading**

What is a method heading? *Look up* **method heading** to find "the name of a method, its **return type**, followed by a **parameter list** in parentheses."

What is a parameter list? *Look up* **parameter list** to find "a **list** of **parameters**." *Look up* **list**, *look up* **parameters**, and *look up* **return type**, and you finally get a definition of all of the terms using the same method you used to *look up* the original term. And then, when all new terms are defined, you have a definition for **object**.

Recursion Example 2

A definition of a *queue*:

empty
or has one element at the front of the queue followed by a *queue*

Recursion Example 3

An arithmetic expression is defined as one of these:

a numeric constant such as 123 or –0.001
or a numeric variable that stores a numeric constant
or an *arithmetic expression* enclosed in parentheses
or an *arithmetic expression* followed by a binary operator (`+`, `-`, `/`, `%`, or `*`)
followed by an *arithmetic expression*

Characteristics of Recursion

A **recursive definition** is a definition that includes a simpler version of itself. One example of a recursive definition is given next: the power method that raises an integer (x) to an integer power (n). This definition is recursive because x^{n-1} is part of the definition itself. For example,

$$4^3 = 4 \times 4^{(n-1)} = 4 \times 4^{(3-1)} = 4 \times 4^2$$

What is 4^2? Using the recursive definition above, 4^2 is defined as:

$$4^2 = 4 \times 4^{(n-1)} = 4 \times 4^{(2-1)} = 4 \times 4^1$$

and 4^1 is defined as

$$4^1 = 4 \times 4^{(n-1)} = 4 \times 4^{(1-1)} = 4 \times 4^0$$

and 4^0 is a base case defined as

$$4^0 = 1$$

The recursive definition of 4^3 includes 3 recursive definitions. The base case is `n==0`:
$x^n = 1$ if $n = 0$

To get the actual value of 4^3, work backward and let 1 replace 4^0, 4 * 1 replace 4^1, 4 * 4^1 replace 4^2, and 4 * 4^2 replace 4^3. Therefore, 4^3 is defined as 64.

To be recursive, an algorithm or method requires at least one recursive case and at least one base case. The recursive algorithm for power illustrates the characteristics of a recursive solution to a problem.

- The problem can be decomposed into a simpler version of itself in order to bring the problem closer to a base case.
- There is at least one base case that does not make a recursive call.
- The partial solutions are managed in such a way that all occurrences of the recursive and base cases can communicate their partial solutions to the proper locations (values are returned).

Comparing Iterative and Recursive Solutions

For many problems involving repetition, a recursive solution exists. For example, an iterative solution is shown below along with a recursive solution in the `TestPowFunctions` class, with the methods `powLoop` and `powRecurse`, respectively. First, a unit test shows calls to both methods, with the same arguments and same expected results.

```java
// File RecursiveMethodsTest.java
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class RecursiveMethodsTest {

  @Test
  public void testPowLoop() {
    RecursiveMethods rf = new RecursiveMethods();
    assertEquals(1, rf.powLoop(4, 0));
    assertEquals(1, rf.powRecurse(4, 0));

    assertEquals(4, rf.powLoop(4, 1));
    assertEquals(4, rf.powRecurse(4, 1));

    assertEquals(16, rf.powLoop(2, 4));
    assertEquals(16, rf.powRecurse(2, 4));
  }
}
```

```java
// File RecursiveMethods.java
public class RecursiveMethods {

  public int powLoop(int base, int power) {
    int result;
    if (power == 0)
      result = 1;
    else {
      result = base;
      for (int j = 2; j <= power; j++)
        result = result * base;
    }
    return result;
  }

  public int powRecurse(int base, int power) {
    if (power == 0)
      return 1;
    else
                    // Make a recursive call \\
      return base * powRecurse(base, power - 1);
  }
}
```

In `powRecurse`, if n is 0—the base case—the method call evaluates to 1. When n > 0—the recursive case—the method is invoked again with the argument reduced by one. For example, `powRecurse(4,1)` calls `powRecurse(4, 1-1)`, which immediately returns 1.

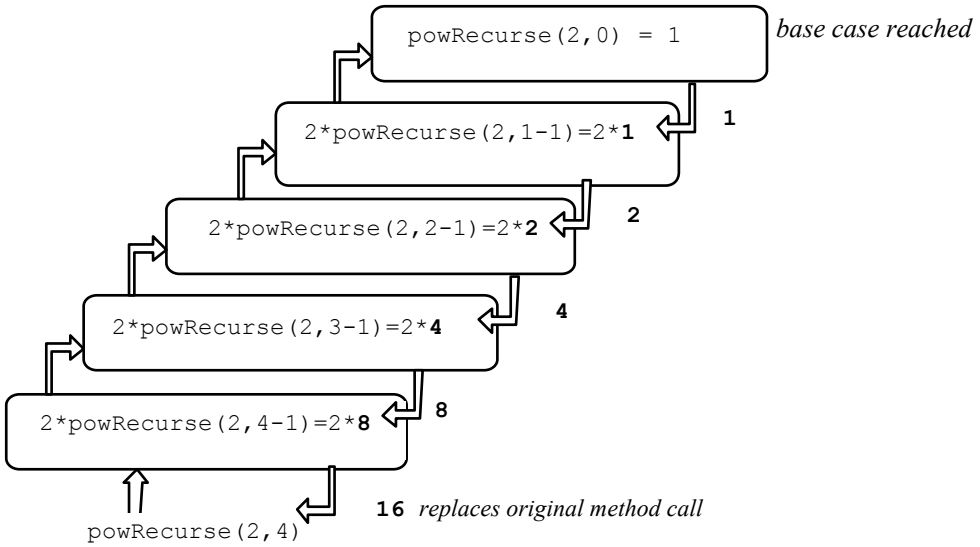
For another example, the original call `powRecurse(2,4)` calls `powRecurse(2,3)`, which then calls `powRecurse(2,2)`, which then calls `powRecurse(2,1)`, which then calls `powRecurse(2,0)`, which returns 1.

Then, `2*powRecurse(2,0)`evaluates to 2*1, or 2, so `2*powRecurse(2,1)` evaluates to 4, `2*powRecurse(2,2)` evaluates to 8, `2*powRecurse(2,3)` evaluates to 16, and `2*powRecurse(2,4)` evaluates to 32.

Tracing recursive methods requires diligence, but it can help you understand what is going on. Consider tracing a call to the recursive power method to get 2^4.

```
assertEquals(16, rf.powRecurse(2, 4));
```

After the initial call to `powRecurse(2,4)`, `powRecurse` calls another instance of itself until the base case of `power==0` is reached. The following picture illustrates a method that calls instances of the same method. The arrows that go up indicate this. When an instance of the method can return something, it returns that value to the method that called it. The arrows that go down with the return values written to the right indicate this.



The final value of 16 is returned to the `main` method, where the arguments of 2 and 4 were passed to the first instance of `powRecurse`.

Self-Check

17-1 What is the value of `rf.powRecurse(3, 0)`?

17-2 What is the value of `rf.powRecurse(3, 1)`?

17-3 Fill in the blanks with a trace of the call `rf.powRecurse(3, 4)`

Tracing Recursive Methods

In order to fully understand how recursive tracing is done, consider a series of method calls given the following method headers and the simple `main` method:

```
// A program to call some recursive methods
public class Call2RecursiveMethods {

  public static void main(String[] args) {
    Methods m = new Methods();
    System.out.println ("Hello");
    m.methodOne(3);
    m.methodTwo(6);
  }
}

// A class to help demonstrate recursive method calls
public class Methods {
  public void methodOne(int x) {
    System.out.println("In methodOne, argument is " + x);
  }

  public void methodTwo(int z) {
    System.out.println("In methodTwo, argument is " + z);
  }
}
```

Output

```
Hello
In methodOne, argument is 3
In methodTwo, argument is 6
```

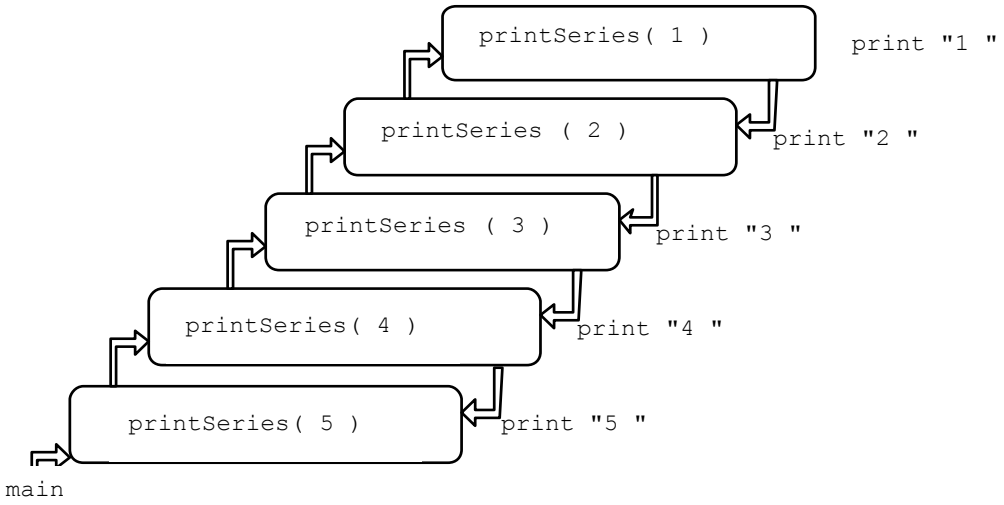
This program begins by printing out `"Hello"`. Then a method call is made to `methodOne`. Program control transfers to `methodOne`, but not before remembering where to return to. After pausing execution of `main`, it begins to execute the body of the `methodOne` method. After `methodOne` has finished, the program flow of control goes back to the last place it was and starts executing where it left off—in this case, in the `main` method, just after the call to `methodOne` and just before the call to `methodTwo`. Similarly, the computer continues executing main and again transfers control to another method: `methodTwo`. After it completes execution of `methodTwo`, the program terminates.

The above example shows that program control can go off to execute code in other methods and then know the place to come back to. It is relatively easy to follow control if no recursion is involved. However, it can be difficult to trace through code with recursive methods. Without recursion, you can follow the code from the beginning of the method to the end. In a recursive method, you must trace through the same method while trying to remember how many times the method was called, where to continue tracing, and the values of the local variables (such as the parameter values). Take for example the following code to print out a list of numbers from 1 to n, given n as an input parameter.

```
  public void printSeries(int n) {
    if (n == 1)
      System.out.print(n + " ");
    else  {
      printSeries(n - 1);
      // after recursive call \\
      System.out.print(n + " ");
    }
  }
```

A call to `printSeries(5)` generates the output: `1 2 3 4 5`

Let's examine step by step how the result is printed. Each time the method is called, it is stacked with its argument. For each recursive case, the argument is an integer one less than the previous call. This brings the method one step closer to the base case. When the base case is reached (`n==1`) the value of `n` is printed. Then the previous method finishes up by returning to the last line of code below `/* after recursive call */`.

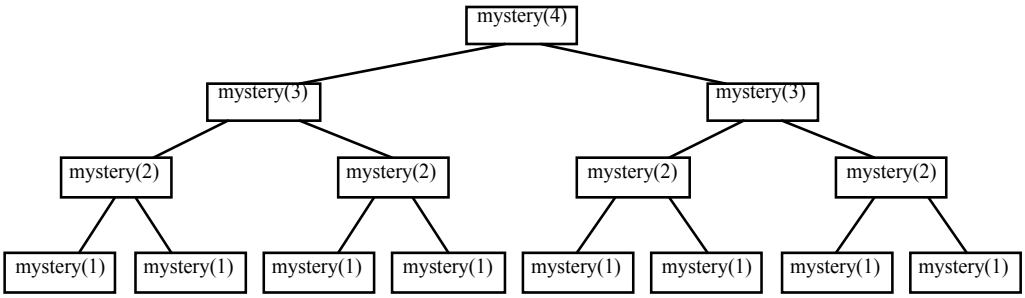


Recursive execution of printSeries(5)

Notice that when a new recursive call is made, the current invocation of the method `printSeries(5)` starts a completely new invocation of the same method `printSeries(4)`. The system pushes the new method invocation on the top of a stack. Method invocations are pushed until the method finds a base case and finishes. Control returns back to previous invocation `printSeries(2)` by popping the stack, and the value of `n` (`2`) prints. Now consider a method that has multiple recursive calls within the recursive case.

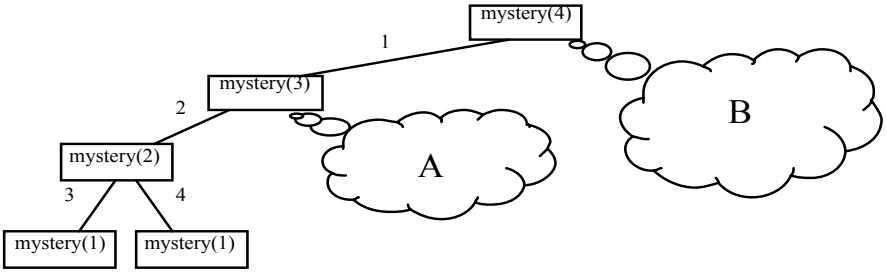
```
public void mystery(int n) {
  if (n == 1)
    System.out.print(" 1 ");
  else {
    mystery(n - 1);
    System.out.print("<" + n + ">");
    mystery(n - 1);
  }
}
```

When the base case has not yet been reached, there is a recursive call, then a print statement, and then another recursive call. With two recursive calls, it proves more insightful to approach a trace from a graphical perspective. A method call tree for `mystery(4)` looks like this.



Recursive execution of mystery(4)

As you can see, when there are multiple recursive calls in the same method, the number of calls increases exponentially — there are eight calls to `mystery(1)`. The recursion reaches the base case when at the lowest level of the structure (at the many calls to `mystery(1)`). At that point `" 1 "` is printed out and control returns to the calling method. When the recursive call returns, `"<" + n + ">"` is printed and the next recursive call is called. First consider the left side of this tree. The branches that are numbered 1, 2, 3, and 4 represent the method calls after `mystery(4)` is called. After the call #3 to `mystery`, `n` is `1` and the first output `" 1 "` occurs.



The first part of mystery

Control returns to the previous call when `n` was 2. `<2>` is printed. The output so far:

```
1 <2>
```

Then a recursive call is a made as `mystery(2-1)` and `" 1 "` is printed again. The output so far:

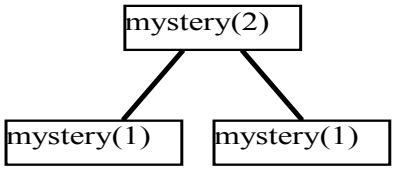
```
1 <2> 1
```

Control then returns to the first call to mystery(3) and <3> is printed. The output so far:

```
1 <2> 1 <3>
```

Then these method calls behind

occur.



With n == 2, the base case is skipped and the recursive case is called once again. This means another call to `mystery(2-1)`, which is `" 1 "`, a printing of `<2>`, followed by another call to `mystery(2-1)`, which is yet another `" 1 "`. Add these three prints to the previous output and the output so far is:

```
1 <2> 1 <3> 1 <2> 1
```

This represents the output from `mystery(3)`. Control then returns to the original call `mystery(4)` when `<4>` is printed. Then the cloud behind B prints the same output as `mystery(3)`, the output shown immediately above. The final output is `mystery(3)`, followed by printing `n` when `n` is 4, followed by another mystery(3).

```
1 <2> 1 <3> 1 <2> 1 <4> 1 <2> 1 <3> 1 <2> 1
```

Self-Check

17-4 Describe the output that would be generated by the message `mystery(5);` .

17-5 What does `mystery2(4)` return?

```
public void mystery2(int n)  {
  if (n > 1)
    mystery2(n - 1);
  System.out.print(n + " ");
}
```

Infinite Recursion

Infinite recursion occurs when a method keeps calling other instances of the same method without making progress towards the base case or when the base case can never be met.

```
public int sum(int n) {
  if (n  ==  0)
    return 0;
  else
    return n + sum(n + 1);
}
```

In this example, no progress is made towards the base case when n is a positive integer because every time sum is called, it is called with a larger value, so the base condition will never be met.

Recursion and Method Calls

Recursion can be implemented on computer systems by allowing each method call to create a **stack frame** (also known as an activation record). This stack frame contains the information necessary for the proper execution of the many methods that are active while programs run. Stack frames contain information about the values of local variables, parameters, the return value (for non-void methods), and the return address where the program should continue executing after the method completes. This approach to handling recursive method calls applies to all methods. A recursive method does not call itself; instead, a recursive call creates an instance of a method that just happens to have the same name.

With or without recursion, there may be one too many stack frames on the stack. Each time a method is called, memory is allocated to create the stack frame. If there are many method calls, the computer may not have enough memory. Your program could throw a

`StackOverflowError`. In fact you will get a `StackOverflowError` if your recursive case does not get you closer to a base case.

```
// Recursive case does not bring the problem closer to the base case.
public int pow(int base, int power) {
  if (power == 0)
    return 1;
  else
    return base * pow(base, power + 1); // <- should be power - 1
}
```

```
java.lang.StackOverflowError
```

The exception name hints at the fact that the method calls are being pushed onto a stack (as stack frames). At some point, the capacity of the stack used to store stack frames was exceeded. `pow`, as written above, will never stop on its own.

Self-Check

17-6 Write the return value of each.

a. ____ `mystery6(-5)`
b. ____ `mystery6(1)`
c. ____ `mystery6(2)`
d. ____ `mystery6(3)`
e. ____ `mystery6(4)`

```
public int mystery6(int n) {
  if (n < 1)
    return 0;
  else if (n == 1)
    return 1;
  else
    return 2 * mystery6(n - 1);
}
```

17-7 Write the return value of each.

a. ____ `mystery7(14, 7)` b. ____ `mystery7(3, 6)` c. ____ `mystery7(4, 8)`

```
public boolean mystery7(int a, int b) {
  if (a >= 10 || b <= 3)
    return false;
  if (a == b)
    return true;
  else
   return mystery7(a + 2, b - 2) || mystery7(a + 3, b - 4);
}
```

17-8 Given the following definition of the Fibonacci sequence, write a recursive method to compute the nth term in the sequence.

```
fibonacci(0) = 1;
fibonacci(1) = 1;
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2); when n >= 2
```

17-9 Write recursive method `howOften` as if it were n class RecursiveMethods that will compute how often a substring occurs in a string. Do not use a loop. Use recursion. These assertions must pass:

```
@Test
public void testSequentialSearchWhenNotHere() {
  RecursiveMethods rm = new RecursiveMethods();
  assertEquals(5, rm.howOften("AAAAA", "A")) ;
  assertEquals(0, rm.howOften("AAAAA", "B")) ;
  assertEquals(2, rm.howOften("catdogcat", "cat")) ;
  assertEquals(1, rm.howOften("catdogcat", "dog")) ;
  assertEquals(2, rm.howOften("AAAAA", "AA")) ;
}
```

17.2 Recursive Palindrome Checker

Suppose that you had a word and you wanted the computer to check whether or not it was a palindrome. A **palindrome** is a word that is the same whether read forward or backward; radar, madam, and racecar, for example. To determine if a word is a palindrome, you could put one finger under the first letter, and one finger under the last letter. If those letters match, move your fingers one letter closer to each other, and check those letters. Repeat this until two letters do not match or your fingers touch because there are no more letters to consider.

The recursive solution is similar to this. To solve the problem using a simpler version of the problem, you can check the two letters on the end. If they match, ask whether the `String` with the end letters removed is a palindrome.

The base case occurs when the method finds a `String` of length two with the same two letters. A simpler case would be a `String` with only one letter, or a `String` with no letters. Checking for a `String` with 0 or 1 letters is easier than comparing the ends of a `String` with the same two letters. When thinking about a base case, ask yourself, "Is this the simplest case? Or can I get anything simpler?" Two base cases (the number of characters is 0 or 1) can be handled like this (assume `str` is the `String` object being checked).

```
if (str.length() <= 1)
  return true;
```

Another base case is the discovery that the two end letters are different when `str` has two or more letters.

```
else if (str.charAt(0) != str.charAt(str.length() - 1))
  return false; // The end characters do not match
```

So now the method can handle the base cases with `String`s such as `""`, `"A"`, and `"no"`. The first two are palindromes; `"no"` is not.

If a `String` is two or more characters in length and the end characters match, no decision can be made other than to keep trying. The same method can now be asked to solve a simpler version of the problem. Take off the end characters and check to see if the smaller string is a palindrome. `String`'s `substring` method will take the `substring` of a `String` like `"abba"` to get `"bb"`.

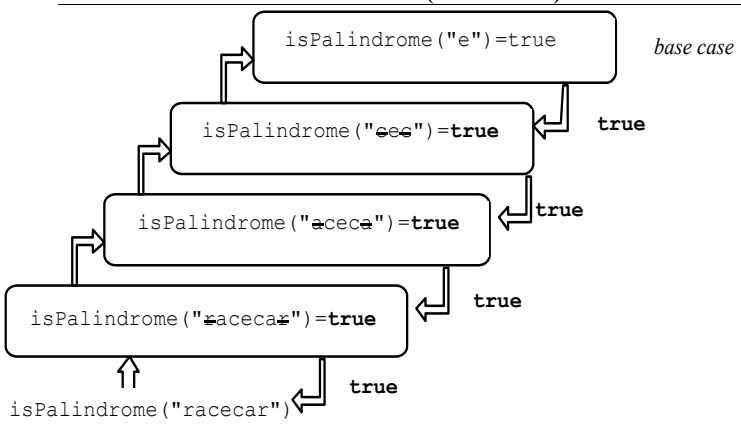
```
// This is a substring of the original string
// with both end characters removed.
return isPalindrome(str.substring(1, str.length() - 1));
```

This message will not resolve on the next call. When `str` is `"bb"`, the next call is `isPalindrome("")`, which returns `true`. It has reached a base case—length is 0. Here is a complete recursive palindrome method.

```
// Return true if str is a palindrome or false if it is not
public boolean isPalindrome(String str) {
  if (str.length() <= 1) {
    // Base case when this method knows to return true.
    return true;
  }
  else if (str.charAt(0) != str.charAt(str.length() - 1)) {
    // Base case when this method knows to return false
    // because the first and last characters do not match.
    return false;
  }
  else {
    // The first and last characters are equal so check if the shorter
    // string--a simpler version of this problem--is a palindrome.
    return isPalindrome(str.substring(1, str.length() - 1));
  }
}
```

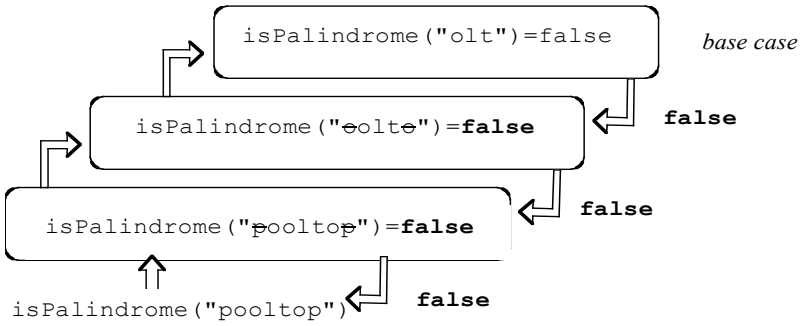
If the length of the string is greater than 1 and the end characters match, `isPalindrome` calls another instance of `isPalindrome` with smaller and smaller `String` arguments until one base case is reached. Either a `String` is found that has a length less than or equal to 1, or the characters on the ends are not the same. The following trace of `isPalindrome("racecar")` visualizes the calls that are made in this way.

`isPalindrome` Recursive Calls (true result)



Since the fourth (topmost) call to `isPalindrome` is called with the `String` `"e"`, a base case is found—a `String` with length 1. This `true` value gets returned to its caller (argument was `"e"`), which in turn returns `true` back to its caller (the argument was `"cec"`), until `true` gets passed back to the first caller of `isPalindrome`, the method call with the original argument of `"racecar"`, which returns the value `true`. Now consider tracing the recursive calls for the `String` `"pooltop"`.

`isPalindrome` Recursive Calls (false result)



The base case is reached when the method compares the letters at the ends—`"o"` and `"t"` do not match. That particular method call returns `false` back to its caller (whose argument was `"oolto"`), which returns `false` to its caller. The original call to `isPalindrome("pooltop")` is replaced with `false` to the method that originally asked if `"pooltop"` was a palindrome.

Self-Check

17-10 What value is returned from `isPalindrome("yoy")`?

17-11 What value is returned from `isPalindrome("yoyo")`?

17-12 Write the return value of each method call

a. _____ `huh("+abc+");`
b. _____ `huh("-abc-");`
c. _____ `huh("-a-b-c-");`
d. _____ `huh("------abc-----");`

```
public String huh(String str) {
  if (str.charAt(0) == '-')
    return huh(str.substring(1, str.length()));
  else if (str.charAt(str.length() - 1) == '-')
    return huh(str.substring(0, str.length() - 1));
  else
    return str;
}
```

17.3 Recursion with Arrays

The *sequential search* algorithm uses an integer subscript that increases if an element is not found and the index is still in the range (meaning that there are more elements to compare). This test method demonstrates the desired behavior.

```
@Test
public void testSequentialSearchWhenHere() {
  RecursiveMethods rm = new RecursiveMethods();
  String[] array = { "Kelly", "Mike", "Jen", "Marty", "Grant" };
  int lastIndex = array.length - 1;

  assertTrue(rm.exists(array, lastIndex, "Kelly"));
  assertTrue(rm.exists(array, lastIndex, "Mike"));
  assertTrue(rm.exists(array, lastIndex, "Jen"));
  assertTrue(rm.exists(array, lastIndex, "Marty"));
  assertTrue(rm.exists(array, lastIndex, "Grant"));
}
```

The same algorithm can be implemented in a recursive fashion. The two base cases are:

1. If the element is found, return true.
2. If the index is out of range, terminate the search by returning false.

The recursive case looks in the portion of the array that has not been searched. With sequential search, it does not matter if the array is searched from the smallest index to the largest or the largest index to the smallest. The `exists` message compares the search element with the largest valid array index. If it does not match, the next call narrows the search. This happens when the recursive call simplifies the problem by decreasing the index. If the element does not exist in the array, eventually the index goes to -1 and the method returns `false` to the preceding call, which returns `false` to the preceding call, until the original method call to `exists` returns `false` to the point of the call.

```
// This is the only example of a parameterized method.
// The extra <T>s allow any type of arguments.
public <T> boolean exists(T[] array, int lastIndex, T target) {
  if (lastIndex < 0) {
    // Base case 1: Nothing left to search
    return false;
  } else if (array[lastIndex].equals(target)) { // Base case 2: Found it
    return true;
  } else { // Recursive case
    return exists(array, lastIndex - 1, target);
  }
}
```

A test should also be made to ensure exists returns false when the target is not in the array.

```
@Test
public void testSequentialSearchWhenNotHere() {
  RecursiveMethods rm = new RecursiveMethods();
  Integer[] array = { 1, 2, 3, 4, 5 };
  int lastIndex = array.length - 1;
  assertFalse(rm.exists(array, lastIndex, -123));
  assertFalse(rm.exists(array, lastIndex, 999));
}
```

Self-Check

17-13 What would happen when `lastIndex` is not less than the array's capacity as in this assertion?

```
assertFalse(rm.exists(array, array.length + 1, "Kelly"));
```

17-14 What would happen when `lastIndex` is less than 0 as in this assertion?

```
assertFalse(rm.exists(array, -1, "Mike"));
```

17-15 Write a method `printForward` that prints all objects referenced by the array named `x` (that has `n` elements) from the first element to the last. Use recursion. Do not use any loops. Use this method heading:

```
public void printForward(Object[] array, int n)
```

17-16 Complete this `testReverse` method so a method named reverse in class RecursiveMethods will reverse the order of all elements in an array of Objects that has `n` elements. Use this heading:

```
public void printReverse(Object[] array, int leftIndex, int rightIndex)

    @Test
    public void testReverse() {
      RecursiveMethods rm = new RecursiveMethods();

        String[] array =  { "A", "B", "C" };
        rm.reverse(array, 0, 2);
        assertEquals(                        );
        assertEquals(                        );
        assertEquals(                        );
      }
```

17-17 Write the recursive method reverse is if it were in class RecursiveMethods. Use recursion. Do not use any loops.

17.4 Recursion with a Linked Structure

This section considers a problem that you have previously resolved using a loop — searching for an object reference from within a linked structure. Consider the base cases first.

The simplest base case occurs with an empty list. In the code shown below, this occurs when there are no more elements to compare. A recursive find method returns `null` to indicate that the object being searched for did not equal any in the list. The other base case is when the object is found. A recursive method then returns a reference to the element in the node.

The recursive case is also relatively simple: If there is some portion of the list to search (not yet at the end), and the element is not yet found, search the remainder of the list. This is a simpler version of the same problem – search the list that does not have the element that was just compared. In summary, there are two base cases and one recursive case that will search the list beginning at the next node in the list:

Base cases:
If there is no list to search, return `null`.
If the current node equals the object, return the reference to the data.
Recursive case:
Search the remainder of the list – from the next node to the end of the list

The code for a recursive search method is shown next as part of `class SimpleLinkedList`. Notice that there are two `findRecursively` methods — one public and one private. (Two methods of the same name are allowed in one class if the number of parameters differs.) This allows users to search without knowing the internal implementation of the class. The public method requires the object being searched for, but not the private instance variable named `front`, or any knowledge of the `Node` class. The public method calls the private method with the element to search for along with the first node to compare — `front`.

```
public class SimpleLinkedList<E> {

  private class Node  {
    private E data;
    private Node next;

    public Node(E objectReference, Node nextReference) {
      data = objectReference;
      next = nextReference;
    }
  } // end class Node

  private Node front;
```

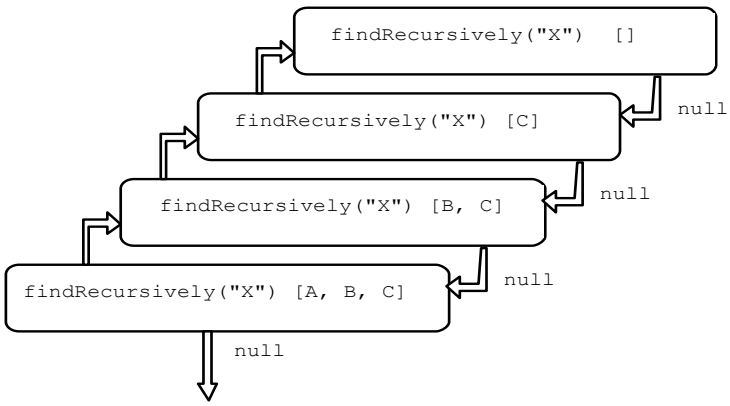
```
  public SimpleLinkedList() {
    front = null;
  }

  public void addFirst(E element) {
    front = new Node(element, front);
  }

  // Return a reference to the element in the list that "equals" target
  // Precondition: target's type overrides "equals" to compare state
  public E findRecursively(E target) {
    // This public method hides internal implementation details
    // such as the name of the reference to the first node to compare.
    //
    // The private recursive find, with two arguments, will do the work.
    // We don't want the programmer to reference first (it's private).
    // Begin the search at the front, even if front is null.
    return findRecursively(target, front);
  }

  private E findRecursively(E target, Node currentNode) {
    if (currentNode == null) // Base case--nothing to search for
      return null;
    else if (target.equals(currentNode.data)) // Base case -- element found
      return currentNode.data;
    else
      // Must be more nodes to search, and still haven't found it;
      // try to find from the next to the last. This could return null.
      return findRecursively(target, currentNode.next);
  }
}
```

Each time the `public` `findRecursively` method is called with the object to find, the `private` `findRecursively` method is called. This private method takes two arguments: the object being searched for and `front` — the reference to the first node in the linked structure. If `front` is `null`, the private `findRecursively` method returns `null` back to the public method `findRecursively` , which in turn returns `null` back to main (where it is printed).

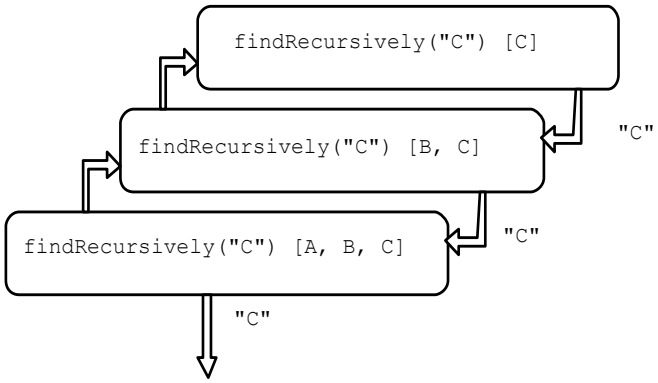
In a non-empty list, `currentNode` refers to a node containing the first element. If the first node's data does not equal the search target, a recursive call is made. The second argument would be a reference to the second node in the list. (The method still needs to pass the object being searched). This time, the problem is simpler because there is one less element in the list to search. Each recursive call to `findRecursively` has a list with one less node to consider. The code is effectively "shrinking" the search area.

The recursive method keeps making recursive calls until there is either nothing left of the list to search (return `null`), or the element being searched for is found in the smaller portion of the list. In this latter case, the method returns the reference back to the public method, which in turn returns the reference back to main (where it is printed).

At some point, the method will eventually reach a base case. There will be one method call on the stack for each method invocation. The worst case for `findRecursively` is the same for sequential search: O(n). This means that a value must be returned to the calling function, perhaps thousands of times. A trace of looking for an element that is not in the list could look like this. The portion of the list being searched is shown to the right of the call (`[ ]` is an empty list):



And here is a trace of a successful search for "C". If "C" were at the end of the list with size() == 975, there would have been 975 method calls on the stack.



Self-Check

17-18 Add recursive method `toString` to the `SimpleLinkedList` class to return a string with the `toString` version of all elements eparated by spaces. Use recursion, do not use a loop.

Answers to Self-Checks

17-1 powRecurse (3, 0) == 1

17-2 powRecurse (3, 1) == 3

17-3 filled in from top to bottom

3*(3, 0) = 1
3*(3, 1) = 3*1 = 3
3*(3, 2) = 3*3 = 9
3*(3, 3) = 3*9 = 27
3*(3, 4) = 3*27 = 81

17-4 result mystery(5)

1 <2> 1 <3> 1 <2> 1 <4> 1 <2> 1 <3> 1 <2> 1 <5> 1 <2> 1 <3> 1 <2> 1 <4> 1 <2> 1 <3> 1 <2> 1

fence post pattern - the brackets follow the numbers being recursed back into the method

17-5 mystery2(4) result: 1 2 3 4

17-6
a. __0__ `mystery6(-5)`
b. __1__ `mystery6(1)`
c. __2__ `mystery6(2)`
d. __4__ `mystery6(3)`
e. __8__ `mystery6(4)`

17-7 a. false b. false c. true

17-8
```
public int fibonacci(int n){
  if(n == 0)
    return 1;
  else if(n ==1)
    return 1;
  else if(n >= 2)
    return fibonacci(n-1) + fibonacci(n-2);
  else
   return -1;
}
```

17-9
```
public int howOften(String str, String sub) {
  int subsStart = str.indexOf(sub);
  if (subsStart < 0)
    return 0;
  else
    return 1 + howOften(str.substring(subsStart + sub.length()), sub);
}
```

17-10 isPalindrome ("yoy") == true

17-11 isPalindrome("yoyo") == false

17-12 return values for huh, in order

a. +abc+
b. abc
c. a-b-c
d. abc

17-13 - if "Kelly" is not found at the first index, it will throw an arrayIndexOutOfBounds exception

17-14 - it will immediately return false without searching

17-15
```
public void printForward(Object[] array, int n) {
  if (n > 0) {
    printForward(array, n - 1);
    System.out.println(array[n-1].toString());
  }
}
```

17-16
```
assertEquals("A", array[2]);
assertEquals("B", array[1]);
assertEquals("C", array[0]);
```

17-17
```
public void reverse(Object[] array, int leftIndex, int rightIndex) {
  if (leftIndex < rightIndex) {
    Object temp = array[leftIndex];
    array[leftIndex] = array[rightIndex];
    array[rightIndex] = temp;
    reverse(array, leftIndex + 1, rightIndex - 1);
  }
}
```

17-18
```
public String toString (){
  return toStringHelper(front);
}

private String toStringHelper(Node ref) {
  if(ref == null)
    return "";
  else
    return ref.data.toString() + " " + toStringHelper(ref.next);
}
```

Chapter 18

Binary Trees

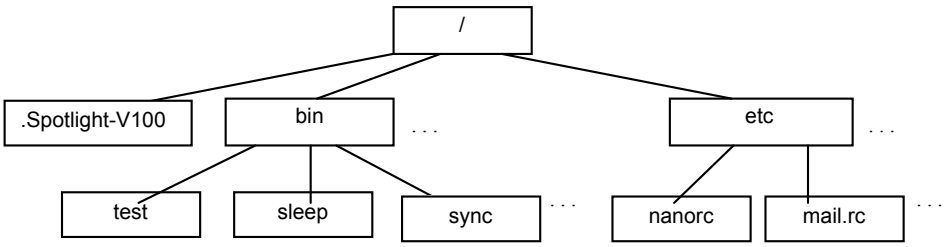
The data structures presented so far are predominantly linear. Every element has one unique predecessor and one unique successor (except the first and last elements). Arrays, and singly linked structures used to implement lists, stacks, and queues all have this linear characteristic. The **tree** structure presented in this chapter is a hierarchical in that nodes may have more than one successor.

Goals

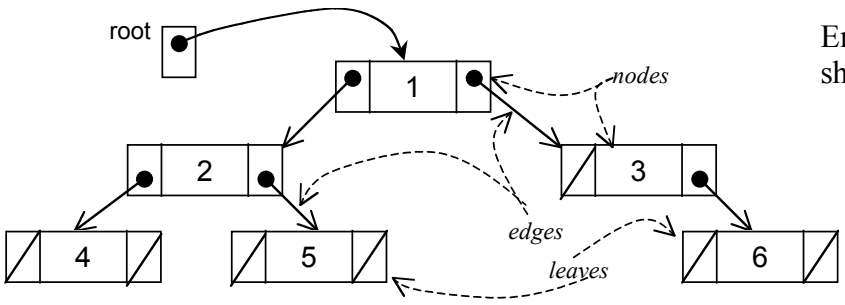
- Become familiar with tree terminology and some uses of trees
- Store data in a hierarchical data structure as a Java Collection class
- Implement binary tree algorithms
- Implement algorithms for a Binary Search Tree

18.1 Trees

Trees are often used to store large collections of data in a hierarchical manner where elements are arranged in successive levels. For example, file systems are implemented as a tree structure with the root directory at the highest level. The collection of files and directories are stored as a tree where a directory may have files and other directories. Trees are hierarchical in nature as illustrated in this view of a very small part of a file system (the root directory is signified as /).



Each node in a tree has exactly one parent except for the distinctive node known as the **root**. Whereas the root of a real tree is usually located in the ground and the leaves are above the root, computer scientists draw trees upside down. This convention allows us to grow trees down from the root since most people find it more natural to write from top to bottom. You are more likely to see the root at the 'top' with the leaves at the 'bottom' of trees. Trees implemented with a linked structure can also be pictured like this:



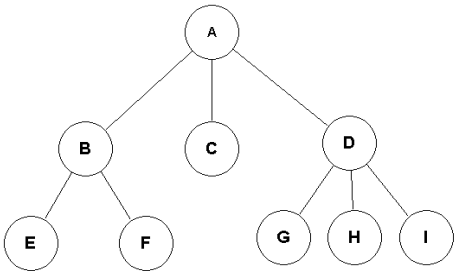
Empty trees are shown here as /

A nonempty tree is a collection of nodes with one node designated as the **root**. Each **node** contains a reference to an element and has **edges** connecting it to other nodes, which are also trees. These other nodes are called children. A tree can be empty — have no nodes. Trees may have nodes with two or more children.

A leaf is a node with no children. In the tree above, the nodes with 4, 5, and 6 are leafs. All nodes that are not leaves are called the internal nodes of a tree, which are 1, 2, and 3 above. A leaf node could later grow a nonempty tree as a child. That leaf node would then become an internal node. Also, an internal node might later have its children become empty trees. That internal node would become a leaf.

A tree with no nodes is called an empty tree. A single node by itself can be considered a tree. A structure formed by taking a node N and one or more separate trees and making N the parent of all roots of the trees is also a tree. This recursive definition enables us to construct trees from existing trees. After the construction, the new tree would contain the old trees as subtrees. A subtree is a tree by itself. By definition, the empty tree can also be considered a subtree of every tree.

All nodes with the same parent are called siblings. The level of a node is the number of edges it takes to reach that particular node from the root. For example, the node in the tree above containing J is at level 2. The height of a tree is the level of the node furthest away from its root. These definitions are summarized with a different tree where the letters A through I represent the elements.

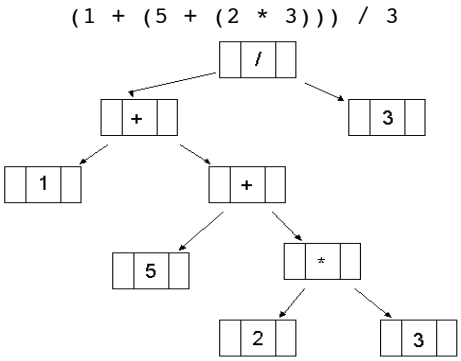


root	A
internal nodes	A B D
leaves	E F C G H I
height	2
level of root	0
level of node with B	1
level of node with E	2
nodes at level 1	3
parent of G, H and I	D
children of B	E F

A **binary tree** is a tree where each node has exactly two binary trees, commonly referred to as the left child and right child. Both the left or right trees are also binary trees. They could be empty trees. When both children are empty trees, the node is considered a leaf. Under good circumstances, binary trees have the property that you can reach any node in the tree within $\log_2 n$ steps, where n is the number of nodes in the tree.

Expression Tree

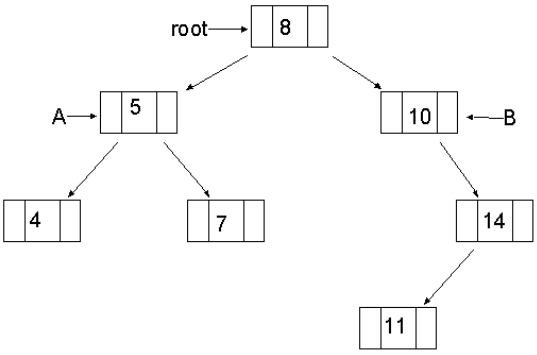
An **expression tree** is a binary tree that stores an arithmetic expression. The tree can then be traversed to evaluate the expression. The following expression is represented as a binary tree with operands as the leaves and operators as internal nodes.



Depending on how you want to traverse this tree — visit each node once — you could come up with different orderings of the same expression: infix, prefix, or postfix. These tree traversal algorithms are presented later in this chapter.

Binary Search Tree

Binary Search Trees are binary trees with the nodes arranged according to a specific ordering property. For example, consider a binary search tree that stores `Integer` elements. At each node, the value in the left child is less than the value of the parent. The right child has a value that is greater than the value of its parent. Also, since the left and right children of every node are binary search trees, the same ordering holds for all nodes. For example, all values in the left subtree will be less than the value in the parent. All values in the right subtree will be greater than the value of the parent.



The left child of the root (referenced by **A**) has a value (5) that is less than the value of the root (8). Likewise, the value of the right child of the root has a value (10) that is greater than the root's value (8). Also, all the values in the subtree referenced by **A** (4, 5, 7), are less than the value in the root (8).

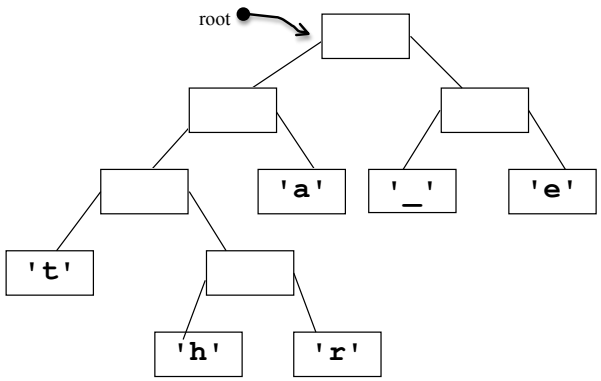
To find the node with the value 10 in a binary search tree, the search begins at the root. If the search value (10) is greater than the element in the root node, search the binary search tree to the right. Since the right tree has the value you are looking for, the search is successful. If the key is further down the tree, the search keeps going left or right until the key is found or the subtree is empty indicating the key was not in the BST. Searching a binary search tree can be O(log n) since

half the nodes are removed from the search at each comparison. Binary search trees store large amounts of real world data because of their fast searching, insertions, and removal capabilities. The binary search tree will be explored later in this chapter.

Huffman Tree

David Huffman designed one of the first compression algorithms in 1952. In general, the more frequently occurring symbols have the shorter encodings. Huffman coding is an integral part of the standards for high definition television (HDTV). The same approach to have the most frequently occurring characters in a text file be represented by shorter codes, allows a file to be compressed to consume less disk space and to take less time to arrive over the Internet.

Part of the compression algorithm involves creation of a Huffman tree that stores all characters in the file as leaves in a tree. The most frequently occurring letters will have the shortest paths in the binary tree. The least occurring characters will have longer paths. For example, assuming a text file contains only the characters 'a', 'e', 'h', 'r', 't', and '_', the Huffman tree could look like this assuming that 'a', 'e', and '_' occur more frequently than 'h' and 'r'.



With the convention that 0 means go left and 1 right, the 6 letters have the following codes:

```
'a'  01
'_'  10
'e'  11
't'  000
'h'  0010
'r'  0011
```

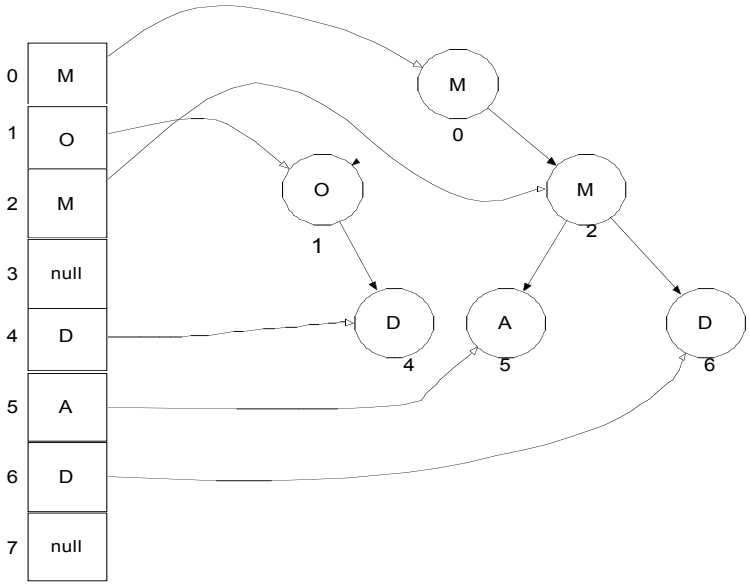
Instead of storing 8 bits for each character, the most frequently occurring letters in this example use only 2 or 3 bits. Some of the characters in a typical file would have codes for some characters that are much longer than 8 bits. These 31 bits represent a text file containing the text "tea_at_three".

```
0001101100100010000010001111 11
  | | | | |  | |  |   |   | | |
  t e a _ a  t _  t   h   r e e
```

Assuming 8 bit ASCII characters, these 31 bits would require 12*8 or 96 bits.

18.2 Implementing Binary Trees

A binary tree can be represented in an array. With an array-based implementation, the root node will always be positioned at array index 0. The root's left child will be positioned at index 1, and the right child will be positioned at array index 2. This basic scheme can be carried out for each successive node counting up by one, and spanning the tree from left to right on a level-wise basis.



Notice that some nodes are not used. These unused array locations show the "holes" in the tree. For example, nodes at indexes 3 and 7 do not appear in the tree and thus have the `null` value in the array. In order to find any left or right child for a node, all that is needed is the node's index. For instance to find node 2's left and right children, use the following formula:

```
Left Child's Index   =  2 * Parent's Index + 1
Right Child's Index  =  2 * Parent's Index + 2
```

So in this case, node 2's left and right children have indexes of 5 and 6 respectively. Another benefit of using an array is that you can quickly find a node's parent with this formula:

```
Parent's Index = (Child's Index – 1) / 2
```

For example, (5-1)/2 and (6-1)/2 both have the same parent in index 2. This works, because with integer division, 4/2 equals 5/2.

Linked Implementation

Binary trees are often implemented as a linked structure. Whereas nodes in a singly linked structure had one reference field to refer to the successor element, a `TreeNode` will have two references — one to the left child and one to the right child. A tree is a collection of nodes with a particular node chosen as the root. Assume the `TreeNode` class will be an inner class with private

instance variables that store these three fields

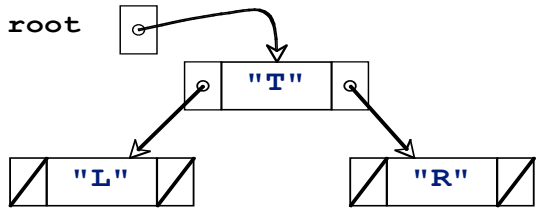
- a reference to the element
- a reference to a left tree (another `TreeNode`),
- a reference to a right tree (another `TreeNode`).

To keep things simple, the `TreeNode` class begins like this so it can store only strings. There are no generics yet.

```
// A type to store an element and a reference to two other TreeNode objects
private class TreeNode {

  private String data;
  private TreeNode left;
  private TreeNode right;

  public TreeNode(String elementReference) {
    data = elementReference;
    left = null;
    right = null;
  }
}
```

The following three lines of code (if in the same class as this inner node class) will generate the binary tree structure shown:

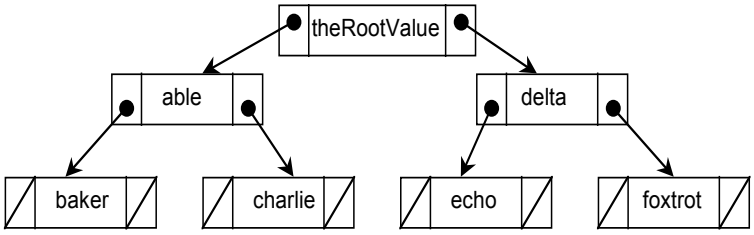
```
TreeNode root = new TreeNode("T");
root.left = new TreeNode("L");
root.right = new TreeNode("R");
```



Self-Check

18-1 Using the tree shown below, identify

a) the root	c) the leaves	e) the children of delta
b) size	d) the internal nodes	f) the number of nodes on level 4



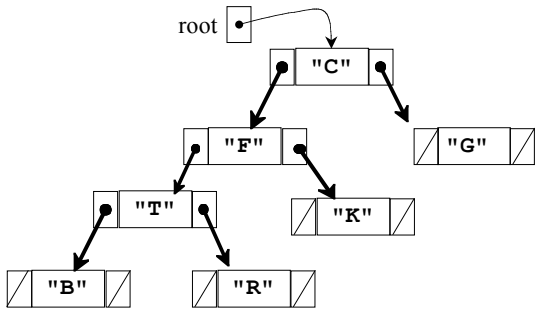
18-2 Using the `TreeNode` class above, write the code that generates the tree above.

Node as an Inner Class

Like the node classes of previous collections, this `TreeNode` class can also be placed inside another. However, instead of a collection class with an insert method, `hardCodeATree` will be used here to create a small binary tree. This will be the tree used to present several binary tree algorithms such as tree traversals in the section that follows.

```
// This simple class stores a collection of strings in a binary tree.
// There is no add or insert method. Instead a tree must be "hard coded" to
// demonstrate algorithms such as tree traversals, makeMirror, and height.
public class BinaryTreeOfStrings {

 private class TreeNode {

   private String data;
   private TreeNode left;
   private TreeNode right;

   public TreeNode(String elementReference) {
     data = elementReference;
      left = null;
      right = null;
    }
  }

  // The entry point into the tree
  private TreeNode root;

  // Construct and empty tree
  public BinaryTreeOfStrings() {
    root = null;
  }

  // Hard code a tree of size 6 on 4 levels
  public void hardCodeATree() {
    root = new TreeNode("C");
    root.left = new TreeNode("F");
    root.left.left = new TreeNode("T");
    root.left.left.left = new TreeNode("B");
    root.left.left.right = new TreeNode("R");
    root.left.right = new TreeNode("K");
    root.right = new TreeNode("G");
  }
}
```

The tree built in `hardCodeATree()`



18.3 Binary Tree Traversals

Code that traverses a linked list would likely visit the nodes in sequence, from the first element to the last. Thus, if the list were sorted in a natural ordering, nodes would be visited in from smallest to largest. With binary trees, the traversal is quite different. We need to stack trees of parents before visiting children. Common tree traversal algorithms include three of a possible six:

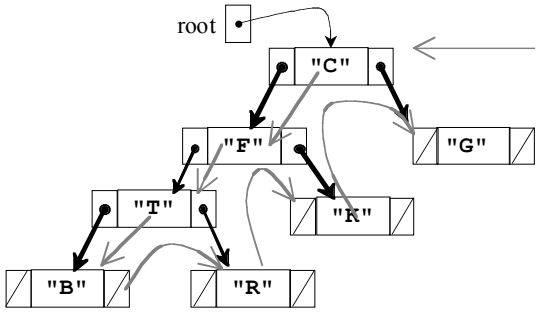
- Preorder: Visit the root, preorder traverse the left tree, preorder traverse the right subtree
- Inorder: Inorder traverse the left subtree, visit the root, inorder traverse the right subtree
- Postorder: Postorder traverse the left subtree, postorder traverse the right, visit the root

When a tree is traversed in a preorder fashion, the parent is processed *before* its children — the left and right subtrees.

Algorithm: Preorder Traversal of a Binary Tree

- Visit the root
- Visit the nodes in the left subtree in preorder
- Visit the nodes in the right subtree preorder

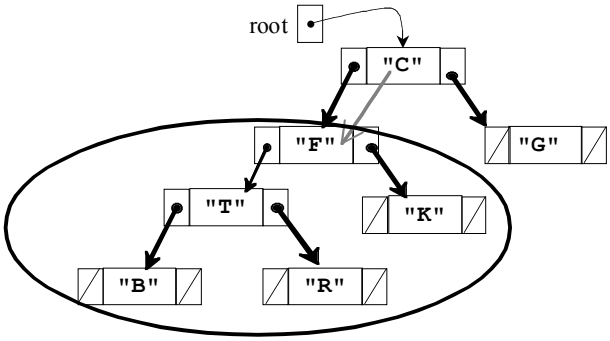
When a binary tree is traversed in a preorder fashion, the root of the tree is "visited" *before* its children — its left and right subtrees. For example, when `preorderPrint` is called with the argument `root`, the element **C** would first be visited. Then a call is made to do a preorder traversal beginning at the left subtree. After the left subtree has been traversed, the algorithm traverses the right subtree of the root node making the element **G** the last one visited during this preorder traversal.



The following method performs a preorder traversal over the tree with "C" in the root node. Writing a solution to this method without recursion would require a stack and a loop. This algorithm is simpler to write with recursion.

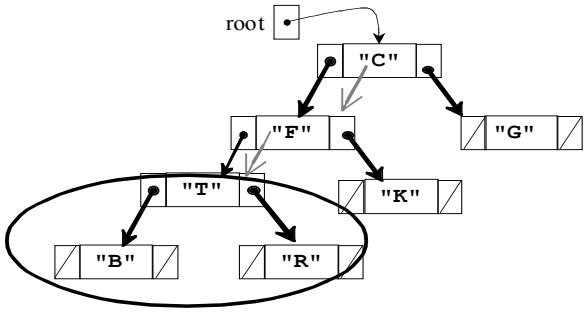
```
public void preOrderPrint() {
  preOrderPrint(root);
}

private void preOrderPrint(TreeNode tree) {
  if (tree != null) {
    // Visit the root
    System.out.print(tree.data + " ");
    // Traverse the left subtree
    preOrderPrint(tree.left);
    // Traverse the right subtree
    preOrderPrint(tree.right);
  }
}
```

When the public method calls `preOrderPrint` passing the reference to the root of the tree, the node with **C** is first visited. Next, a recursive call passes a reference to the left subtree with **F** at the root. Since this TreeNode argument it is not null, **F** is visited next and is printed.



Preorder Traversal so far: **C F**

Next, a recursive call is made with a reference to the left subtree of **F** with **T** at the root, which is visited before the left and right subtrees.

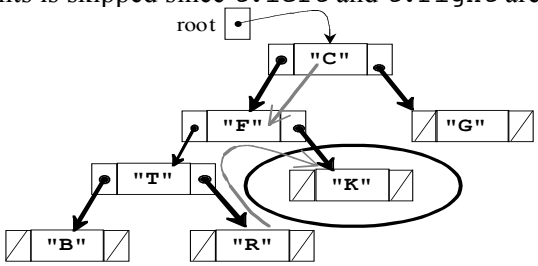


Preorder Traversal so far: C F T

After the root is visited, another recursive call is made with a reference to the left subtree **B** and it is printed. Recursive calls are made with both the left and right subtrees of B. Since they are both null, the if statement is false and the block of three statement is skipped. Control returns to the method with **T** at the root where the right subtree is passed as the argument.

Preorder Traversal so far: C F T B R

The flow of control returns to visiting the right subtree of **F**, which is **K**. The recursive calls are then made for both of K's children (empty trees). Again, in both calls, the block of three statements is skipped since `t.left` and `t.right` are both `null`.



Preorder Traversal so far: C F T B R K

Finally, control returns to visit the right subtree in the first call with the root as the parameter to visit the right subtree in preorder fashion when **G** is printed.

Inorder Traversal

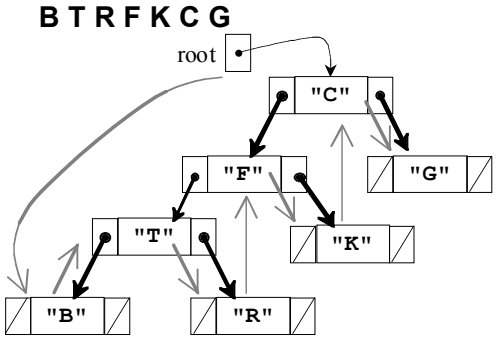
During an inorder traversal, each parent gets processed *between* the processing of its left and right children. The algorithm changes slightly.

- Traverse the nodes in the left subtree inorder
- Process the root
- Traverse the nodes in the right subtree inorder

Inorder traversal visits the root of each tree only after its left subtree has been traversed inorder. The right subtree is traversed inorder after the root.

```
public void inOrderPrint() {
    inOrderPrint(root);
  }

  private void inOrderPrint(TreeNode t) {
    if (t != null) {
      inOrderPrint(t.left);
      System.out.print(t.data + " ");
      inOrderPrint(t.right);
    }
  }
```

Now a call to `inOrderPrint` would print out the values of the following tree as



The `inOrderPrint` method keeps calling `inOrderPrint` recursively with the left subtree. When the left subtree is finally empty, `t.left==null`, the block of three statements executed for **B**.

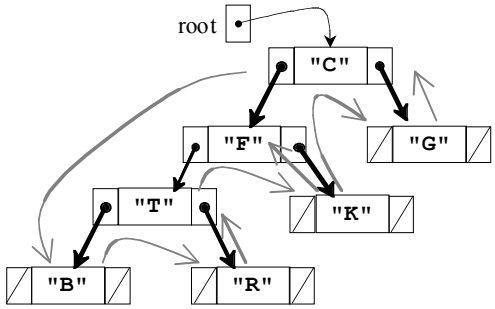
Postorder Traversal

In a postorder traversal, the root node is processed *after* the left and right subtrees. The algorithm shows the process step after the two recursive calls.

1. Traverse the nodes in the left subtree in a postorder manner
2. Traverse the nodes in the right subtree in a postorder manner
3. Process the root

A postorder order traversal would visit the nodes of the same tree in the following fashion:

B R T K F G C



The `toString` method of linear structures, such as lists, is straightforward. Create one big string from the first element to the last. A `toString` method of a tree could be implemented to return the elements concatenated in pre-, in-, or post-order fashion. A more insightful method would be to print the tree to show levels with the root at the leftmost (this only works on trees that are not too big). A tree can be printed sideways with a *reverse* inorder traversal. Visit the right, the root, and then the left.

```
    G
C
        K
    F
            R
        T
            B
```

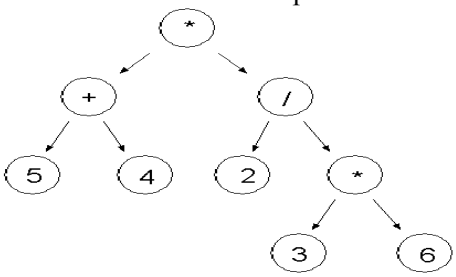
The `printSideways` method below does just this To show the different levels, the additional parameter `depth` begins at 0 to print a specific number of blank spaces depth times before each element is printed. When the root is to be printed depth is 0 and no blanks are printed.

```
public void printSideways() {
  printSideways(root, 0);
}

private void printSideways(TreeNode t, int depth) {
  if (t != null) {
    printSideways(t.right, depth + 1);
    for (int j = 1; j <= depth; j++)
      System.out.print("    ");
    System.out.println(t.data);
    printSideways(t.left, depth + 1);
  }
}
```

Self-Check

18-3 Write out the values of each node of this tree as the tree is traversed both

a. inOrder b. preorder c. postOrder



18-4 Implement the private helper method `postOrderPrint` that will print all elements separated by a space when this public method is called:

```
public void postOrderPrint() {
  postOrderPrint(root);
}
```

18.4 A few other methods

This section provides a few algorithms on binary trees, a few of which you may find useful.

height

The height of an empty tree is -1, the height of an tree with one node (the root node) is 0, and the height of a tree of size greater than 1 is the longest path found in the left tree from the root. The private `height` method first considers the base case to return -1 if the tree is empty.

```
// Return the longest path in this tree or –1 if this tree is empty.
public int height() {
  return height(root);
}

private int height(TreeNode t) {
  if (t == null)
    return -1;
  else
    return 1 + Math.max(height(t.left), height(t.right));
}
```

When there is one node, `height` returns 1 + the maximum height of the left or right trees. Since both are empty, `Math.max` returns -1 and the final result is (1 + -1) or 0. For larger trees, `height` returns the larger of the height of the left subtree or the height of the right subtree.

leafs

Traversal algorithms allow all nodes in a binary tree to be visited. So the same pattern can be used to search for elements, send messages to all elements, or count the number of nodes. In these situations, the entire binary tree will be traversed.

The following methods return the number of leafs in a tree When a leaf is found, the method returns 1 + all leafs to the left + all leafs to the right. If `t` references an internal node (not a leaf), the recursive calls to the left and right must still be made to search further down the tree for leafs.

```
public int leafs() {
  return leafs(root);
}

private int leafs(TreeNode t) {
  if (t == null)
    return 0;
  else {
    int result = 0;
    if (t.left == null && t.right == null)
      result = 1;
    return result + leafs(t.left) + leafs(t.right);
  }
}
```

findMin

The `findMin` method returns the string that precedes all others alphabetically. It uses a preorder traversal to visit the root nodes first (`findMin` could also use be a postorder or inorder traversal). This example show that it may be easier to understand or implement a binary tree algorithm that has an instance variable initialized in the public method and adjusted in the private helper method.

```
// This instance variable is initialized it in the public method findMin.
private String min;

// Return a reference the String that alphabetically precedes all others
public String findMin() {
  if (root == null)
    return null;
  else {
    min = root.data;
    findMinHelper(root);
    return min;
  }
}

public void findMinHelper(TreeNode t) {
  // Only compare elements in nonempty nodes
  if (t != null) {
   // Use a preorder traversal to compare all elements in the tree.
   if (t.data.compareTo(min) < 0)
      min = t.data;
    findMinHelper(t.left);
    findMinHelper(t.right);
  }
}
```

Self-Check

18-5 To `BinaryTreeOfStrings`, add method `findMax` that returns the string that follows all others alphabetically.

18-6 To `BinaryTreeOfStrings`, add method `size` that returns the number of nodes in the tree.

Answers to Self-Checks

18-1 a) theRootValue b) 7 c) baker, Charlie, echo, foxtrot d) theRootValue, able, delta e) echo, foxtrot f) 0 bottom level is 2

18-2 *Assume this code is in a method of class BinaryTreeOfStrings*

```
root = new TreeNode("theRootValue");
root.left = new TreeNode("able");
root.left.left = new TreeNode("baker");
root.left.right = new TreeNode("charlie");
root.right = new TreeNode("delta");
root.right.left = new TreeNode("echo");
root.right.right = new TreeNode("foxtrot");
```

18-3 a. inorder: 5 + 4 * 2 / 3 * 6 b. preorder: * + 5 4 / 2 * 3 6 c. postOrder: 5 4 + 2 3 6 * / *

18-4 *Output using the tree would be:* `baker Charlie able echo foxtrot delta theRootValue`

```
private void postOrderPrint(TreeNode t) {
  if (t != null) {
    postOrderPrint(t.left);
    postOrderPrint(t.right);
    System.out.print(t.data + " ");
  }
}
```

18-5 *This solution shows an extra instance variable can make it easier to understand*

```
private int max;

public int findMax(TreeNode<Integer> t) {
  if (t == null)
    return 0;
  else {
    max = t.data;
    findMaxHelper(t);
    return max;
  }
}

public void findMaxHelper(TreeNode<Integer> t) {
  if (t != null) {
    int temp = ((Integer) t.data).intValue();
    if (temp > max)
      min = temp;
    findMaxHelper(t.left);
    findMaxHelper(t.right);
  }
}
```

18-6

```
public int size() {
  return size(root);
}

private int size(TreeNode t) {
  if (t == null)
    return 0;
  else
    return 1 + size(t.left) + size(t.right);
}
```

Chapter 19

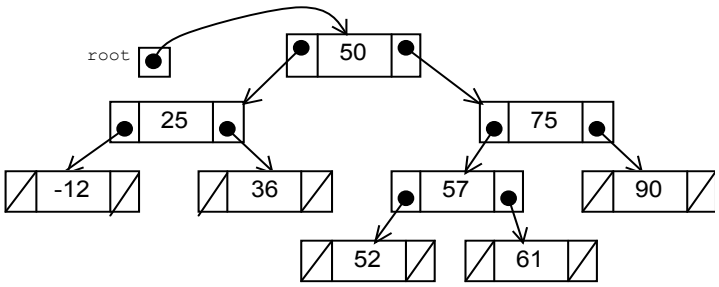
Binary Search Trees

Binary Search Trees

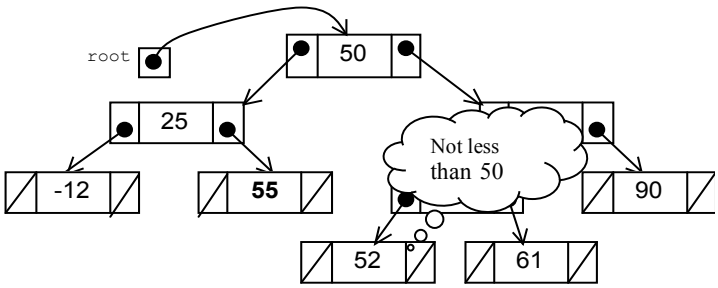
A Binary Search Tree is a binary tree with an ordering property that allows O(log n) insertion, retrieval, and removal of individual elements. Defined recursively, a binary search tree is

1. an empty tree, or
2. consists of a node called the root, and two children, left and right, each of which are themselves binary search trees. Each node contains data at the root that is greater than all values in the left subtree while also being less than all values in the right subtree. No two nodes compare equally. This is called the binary search tree ordering property.

The following two trees represent a binary search tree and a binary tree respectively. Both have the same structure — every node has two children (which may be empty trees shown with `/`). Only the first tree has the binary search ordering property.

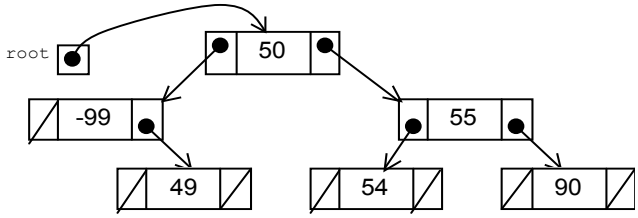


The binary tree below does not have the BST ordering property. The node containing 55 is found in the left subtree of 50 instead of the right subtree.



Self Check

19-1 Is this a binary search tree?



The `BinarySearchTree` class will have the following methods:

- `insert` Add an element to the binary search tree while maintaining the ordering property
- `find` Return a reference to the element that "equals" the argument according to `compareTo`
- `remove` Remove the that "equals" while maintaining the ordering property

Java generics will make this collection class more type safe. It would be tempting to use this familiar class heading.

```
public class BinarySearchTree<E>
```

However, to maintain the ordering property, BinarySearchTree algorithms frequently need to compare two elements to see if one element is greater than, less than, or equal to another element. These comparisons can be made for types that have the compareTo method.

Java generics have a way to ensure that a type has the compareTo method. Rather than accepting any type with <E>, programmers can ensure that the type used to construct an instance does indeed implement the Comparable interface (or any interface that extends the Comparable interface) with this syntax:

```
public class BinarySearchTree <E extends Comparable<E>> {
```

This class heading uses a bounded parameter to restrict the types allowed in a `BinarySearchTree` to `Comparables` only. This heading will also avoid the need to cast to `Comparable`. Using `<E extends Comparable <E>>` will also avoid cast exceptions errors at runtime. Instead, an attempt to compile a construction with a `NonComparable` — assuming `NonComparable` is a class that does not implement `Comparable` — results in a more preferable compile time error.

```
BinarySearchTree<String> strings = new BinarySearchTree<String>();
BinarySearchTree<Integer> integers = new BinarySearchTree<Integer>();
BinarySearchTree<NonComparable> no = new BinarySearchTree<NonComparable>();
                    ⇑
```

Bound mismatch: The type NonComparable is not a valid substitute for the bounded parameter <E extends Comparable<E>>

So far, most elements have been String or Integer objects. This makes explanations shorter. For example, it is easier to write stringTree.insert("A"); than accountTree.insert(new BankAccount("Zeke Nathanielson", 150.00)); (and it is also easier for authors to fit short strings and integers in the boxes that represent elements of a tree).

However, collections of only strings or integers are not all that common outside of textbooks. You will more likely need to store real-world data. Then the find method seems more appropriate. For example, you could have a binary search tree that stores BankAccount objects assuming BankAccount implements Comparable. Then the return value from find could be used to update the object in the collection, by sending withdraw, deposit, or getBalance messages.

```
accountCollection.insert(new BankAccount("Mark", 50.00));
accountCollection.insert(new BankAccount("Jeff", 100.00));
accountCollection.insert(new BankAccount("Nathan", 150.00));

// Need to create a dummy object that will "equals" the account in the BST
BankAccount toBeMatched = new BankAccount("Jeff", -999);
BankAccount currentReference = accountCollection.find(toBeMatched);
assertNotNull(currentReference);
assertEquals("Jeff", currentReference.getID());

accountCollection.printSideways();
currentReference.deposit(123.45);

System.out.println("After a deposit for Jeff");
accountCollection.printSideways();
```

Output (Notice that the element with ID Jeff changes):

```
    Nathan $150.00
Mark $50.00
    Jeff $100.00
After a deposit for Jeff
    Nathan $150.00
Mark $50.00
    Jeff $223.45
```

Linked Implementation of a BST

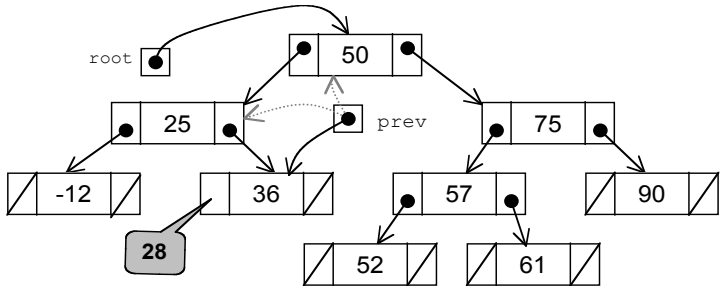
The linked implementation of a binary search tree presented here uses a private inner class `TreeNode` that stores the type `E` specified as the type parameter. This means the nodes can only store the type of element passed as the type argument at construction (which must implement `Comparable` or an interface that extends interface `Comparable`).

```
// This simple class stores a collection of strings in a binary tree.
// There is no add or insert method.Instead a tree will be "hard coded" to
// demonstrate algorithms such as tree traversals, makeMirror, and height.
public class BinarySearchTree<E extends Comparable<E>> {

  private class TreeNode {

    private E data;
    private TreeNode left;
    private TreeNode right;

    TreeNode(E theData) {
      data = theData;
      left = null;
      right = null;
```

```
    }
  }

  private TreeNode root;

  public BinarySearchTree() {
    root = null;
  }

  // The insert and find methods will be added here
}
```

boolean insert(E)

A new node will always be inserted as a leaf. The insert algorithm begins at the root and proceeds as if it were searching for that element. For example, to insert a new `Integer` object with the value of 28 into the following binary search tree, 28 will first be compared to 50. Since 28 is less than the root value of 50, the search proceeds down the left subtree. Since 28 is greater than 25, the search proceeds to the right subtree. Since 28 is less than 36, the search attempts to proceed left, but stops. The tree to the left is empty. At this point, the new element should be added to the tree as the left child of the node with 36.



The search to find the insertion point ends under either of these two conditions:

1. A node matching the new value is found.
2. There is no further place to search. The node can then be added as a leaf.

In the first case, the insert method could simply quit without adding the new node (recall that binary search trees do not allow duplicate elements). If the search stopped due to finding an empty tree, then a new `TreeNode` with the integer 28 gets constructed and the reference to this new node replaces one of the empty trees (the `null` value) in the leaf last visited. In this case, the reference to the new node with 28 replaces the empty tree to the left of 36.

One problem to be resolved is that a reference variable (named `curr` in the code below) used to find the insertion point eventually becomes `null`. The algorithm must determine where it should store the reference to the new node. It will be in either the left link or the right link of the node last visited. In other words, after the insertion spot is found in the loop, the code must determine if the new element is greater than or less than its soon to be parent.

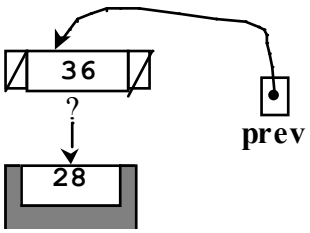
Therefore, two reference variables will be used to search through the binary search tree. The `TreeNode` reference named `prev` will keep track of the previous node visited. (Note: There are other ways to implement this).

The following method is one solution to insertion. It utilizes the Binary Search Tree ordering property. The algorithm checks that the element about to be inserted is either less than or greater than each node visited. This allows the appropriate path to be taken. It ensures that the new element will be inserted into a location that keeps the tree a binary search tree. If the new element

to be inserted compares equally to the object in a node, the insert is abandoned with a `return` statement.

```
public boolean insert(E newElement) {
  // newElement will be added and this will still be a BinarySearchTree.
  // This tree will not insert newElement if it will compareTo an existing
  // element equally.
  if (root == null)
    root = new TreeNode(newElement);
  else {
    // find the proper leaf to attach to
    TreeNode curr = root;
    TreeNode prev = root;
    while (curr != null) {
      prev = curr;
      if (newElement.compareTo(curr.data) < 0)
        curr = curr.left;
      else if (newElement.compareTo(curr.data) > 0)
        curr = curr.right;
      else {
        System.out.println(newElement + " in this BST");
        return false;
      }
    }

    // Determine whether to link the new node came from prev.left or prev.right
    if (newElement.compareTo(prev.data) < 0)
      prev.left = new TreeNode(newElement);
    else
      prev.right = new TreeNode(newElement);
  }
  return true;
}
```

When `curr` finally becomes `null`, it must be from either `prev`'s left or right.



This situation is handled by the code at the end of insert that compares `newElement` to `prev.data`.

Self-Check

19-2 Draw the binary search tree generated by this code:

```
BinarySearchTree<String>  bst = new BinarySearchTree<String>();
bst.insert("Matrix");
bst.insert("Shaft");
bst.insert("Scream");
bst.insert("Titanic");
bst.insert("Antz");
bst.insert("Bean");
```

E find(E)

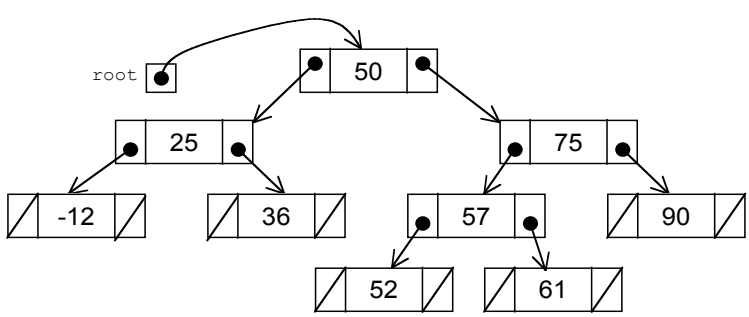
This `BinarySearchTree` needed some way to insert elements before `find` could be tested so insert could be tested, a bit of illogicality. Both will be tested now with a unit test that begins by inserting a small set of integer elements. The `printSideways` message ensures the structure of the tree has the BST ordering property.

```
import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.Before;


public class BinarySearchTreeTest {

  private BinarySearchTree<Integer> aBST;

  // Any test method can use aBST with the same 9 integers shown in @Before as
  // setUpBST will be called before each @Test
  @Before
  public void setUpBST() {
    aBST = new BinarySearchTree<Integer>();
    aBST.insert(50);
    aBST.insert(25);
    aBST.insert(75);
    aBST.insert(-12);
    aBST.insert(36);
    aBST.insert(57);
    aBST.insert(90);
    aBST.insert(52);
    aBST.insert(61);
    aBST.printSideways();
  }
}
```

Output

```
        90
    75
            61
        57
            52
50
        36
    25
        -12
```

Picture the state of aBST



The first test method ensures that elements that can be added result in true and those that can't result in false. Programmers could use this to ensure the element was added or the element already existed.

```
@Test
public void testInsertDoesNotAddExistingElements() {
  assertTrue(aBST.insert(789));
  assertTrue(aBST.insert(-789));
  assertFalse(aBST.insert(50));
  assertFalse(aBST.insert(61));
}
```

This test method ensures that the integers are found and that the correct value is returned.

```
@Test
public void testFindWhenInserted() {
  assertEquals(50, aBST.find(50));
  assertEquals(25, aBST.find(25));
  assertEquals(75, aBST.find(75));
  assertEquals(-12,aBST.find(-12));
  assertEquals(36, aBST.find(36));
  assertEquals(57, aBST.find(57));
  assertEquals(90, aBST.find(90));
  assertEquals(52, aBST.find(52));
  assertEquals(61, aBST.find(61));
}
```

And this test method ensures that a few integers not inserted are also not found.

```
@Test
public void testFindWhenElementsNotInserted() {
  assertNull(aBST.find(999));
  assertNull(aBST.find(0));
}
```

The search through the nodes of a aBST begins at the root of the tree. For example, to search for a node that will compareTo 57 equally, the method first compares 57 to the root element, which has the value of 50. Since 57 is greater than 50, the search proceeds down the *right* subtree (recall that nodes to the right are greater). Then 57 is compared to 75. Since 57 is less than 75, the search proceeds down the *left* subtree of 75. Then 57 is compared to the node with 57. Since these compare equally, a reference to the element is returned to the caller. The binary search continues until one of these two events occur:

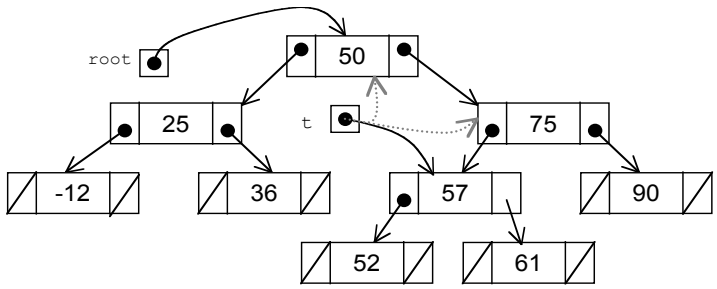
1. The element is found
2. There is an attempt to search an empty tree (nowhere to go—the node is not in the tree)

In the first case, the reference to the data in the node is returned to the sender. In the second case, the method returns null to indicate that the element was not in the tree. Here is an implementation of find method.

```
// Return a reference to the object that will compareTo
// searchElement equally. Otherwise, return null.
public E find(E searchElement) {
  // Begin the search at the root
  TreeNode ref = root;
  // Search until found or null is reached
  while (ref != null) {
    if (searchElement.compareTo(ref.data) == 0)
      return ref.data; // found
    else if (searchElement.compareTo(ref.data) < 0)
      ref = ref.left; // go down the left subtree
```

```
      else
        ref = ref.right; // go down the right subtree
    }
    // Found an empty tree, searchElement was not found.
    return null;
  }
```

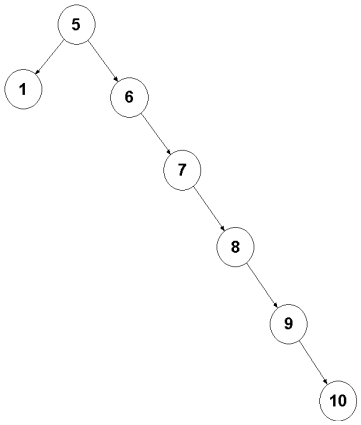
The following picture shows the changing values of the external reference t as it references the three different nodes in its search for 57:



One of the reasons that binary search trees are frequently used to store collections is the speed at which elements can be found. In a manner similar to a binary search algorithm, half of the elements can be eliminated from the search in a BST at each loop iteration. When you go left from one node, you ignore all the elements to the right, which is usually about half of the remaining nodes. Assuming the BinarySearchTree is fairly complete, searching in a binary search tree is O(log n). For example, in the previous search, t referred to only three nodes in a collection of size 9. A tree with 10 levels could have a maximum size of 1,024 nodes. It could take as few as 10 comparisons to find something on level 10.

Efficiency

Much of the motivation for the design of trees comes from the fact that the algorithms are more efficient than those with arrays or linked structures. It makes sense that the basic operations on a binary search tree should require O(log n) time where n is the number of elements of the tree. We know that the height of a balanced binary tree is $\log_2 n$ where *n* is the number elements in the tree. In this case, the cost to find the element should be on the order of O(log *n*). However, with a tree like the following one that is not balanced, runtime performance takes a hit.



If the element we were searching for was the right-most element in this tree (10), the search time would be O(n), the same as a singly linked structure.

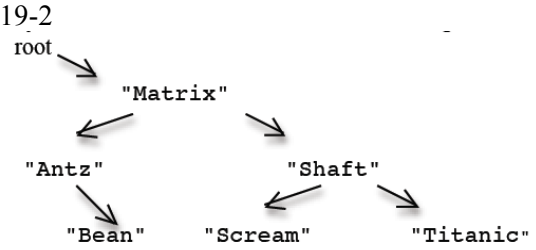
Thus, it is very important that the tree remain balanced. If values are inserted randomly to a binary search tree, this condition may be met, and the tree will remain adequately balanced so that search and insertion time will be O(log *n*).

The study of trees has been very fruitful because of this problem. There are many variants of trees, e.g., red-black trees, AVL trees, B-trees, that solve this problem by re-balancing the tree after operations that unbalance it are performed on them. Re-balancing a binary tree is a very tedious task and is beyond the scope of this book. However, it should be noted that having to rebalance a binary tree every now and then adds overhead to the runtime of a program that requires a binary search tree. But if you are mostly searching, which is often the case, the balanced tree might be appropriate.

The table below compares a binary search tree's performance with a sorted array and singly linked structure (the remove method for BST is left as an exercise).

	Sorted Array	Singly Linked	Binary Search Tree
remove	O(n)	O(n)	O(log n)
find	O(log n)	O(n)	O(log n)
insert	O(n)	O(n)	O(log n)

Answers to Self-Checks

19-1 Yes

19-2