Chapter 1

Program Development

First, there is a need for a computer-based solution to a problem. The need may be expressed in a few sentences like the first examples in this book. The progression from understanding a problem specification to achieving a working computer-based solution is known as "program development."

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	Develop a solution
Implementation	Make the solution run on a computer

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. Emphasis is placed on the deliverables—the tangible results—of each phase. Here is a preview of the deliverables for each of the three stages:

Phase	Deliverable
Analysis	A document that lists the data that store relevant information
Design	An algorithm that outlines a solution
Implementation	An executable program ready to be used by the customer

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.

Analysis Deliverable

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`

Analysis Deliverable

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:

- Problem understood.
- Information to be output: `courseGrade`.
- 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. The following deliverable from the analysis phase summarizes these activities:

Analysis Deliverable

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`

This is the first deliverable. 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 deliverable for the following problem. You will need a calculator to determine the 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 (1) defining an architecture for the program that satisfies the requirements and (2) specifying an algorithm for each program component in the architecture.[1] In later chapters, you will see functions used as the basic building blocks of programs. Then you will see classes used as the basic building blocks of programs. A class is a collection of functions, typically called "methods." In this chapter, the architecture 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.[2] 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 deliverable from the design phase is an algorithm that solves the problem. 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. Obtain `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 deliverable—an algorithm with enough detail to pass on as the input into the next phase, implementation.

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

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

Try to think of program development in terms of the deliverables. This provides a checklist. What deliverables exist so far?

1. From analysis, there is a document with a list of data (variables) and a sample problem.
2. From the design phase there is an algorithm.

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 and hardware to obtain the deliverable. The deliverable of the implementation phase is 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 and associated deliverables:

Activity	Deliverable
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

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`

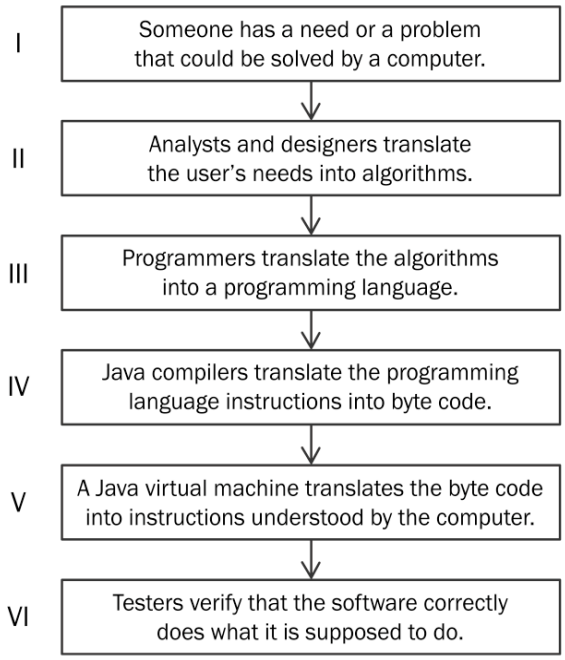
could be translated into Java source code that might look like this:

```
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).

```
// 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 x 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.
public class ShowHello {

  public static void main(String[] args) {
    System.out.println("hello, world");
  }
}
```

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 Chapter 5
arrays Chapters 8-11
interfaces Chapter 12

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;`
-b `anInt = 1.5;`
-c `anInt = "1.5";`
-d `anInt = anInt + 1;`
-e `aDouble = 1;`
-f `aDouble = 1.5;`
-g `aDouble = "1.5";`
-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 and 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 and 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.println("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`
-b `0.5 + x / 2.0`
-c `1.0 + x * 3.0 / y`
-d `1.5 * ( x - y )`
-e `y + -x`
-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.

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

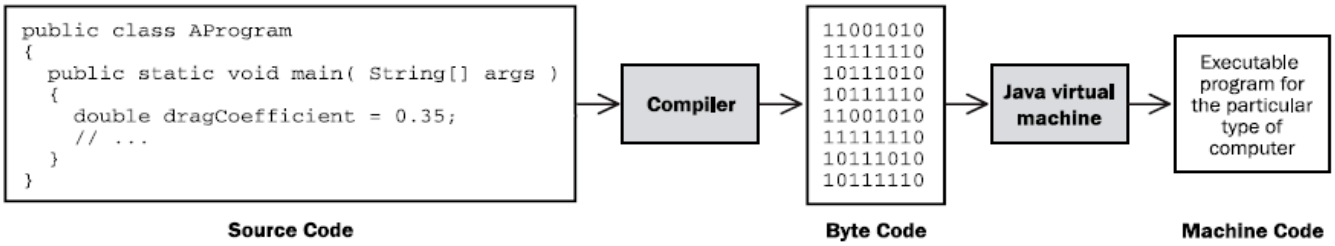
- **syntax errors**—errors that occur when compiling source code into byte code
- **intent errors**—the program does what you typed, not what you intended
- **exception**—errors that occur as the program executes

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 Many Computers



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

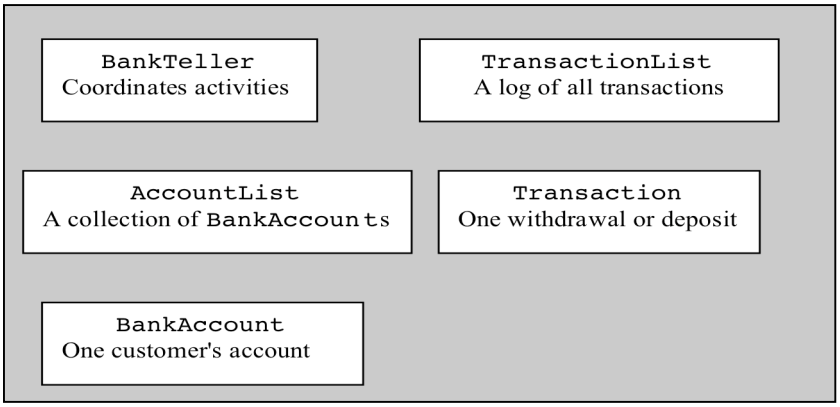
- the problem statement
- an understanding of the problem domain (knowledge of the system that the problem statement may have missed or taken for granted)
- the words spoken during analysis
- the classes that come with the programming language

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 four messages: `getID`, `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 four methods:
  public String getID() {
    return ID;
  }

  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);
```

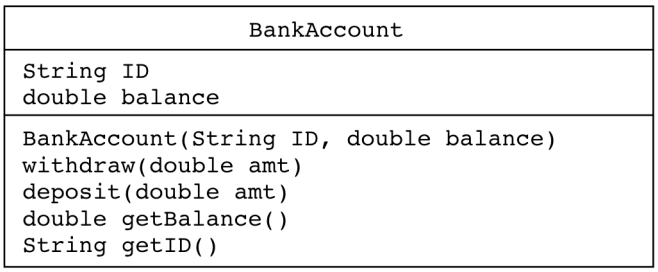
we can derive the following information:

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,

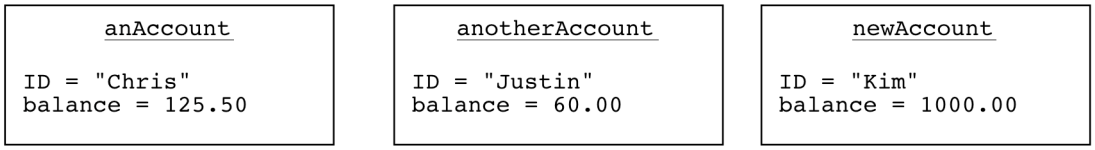
```
BankAccount theNewAccount = new BankAccount("Kim", 1000.00);
```

`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

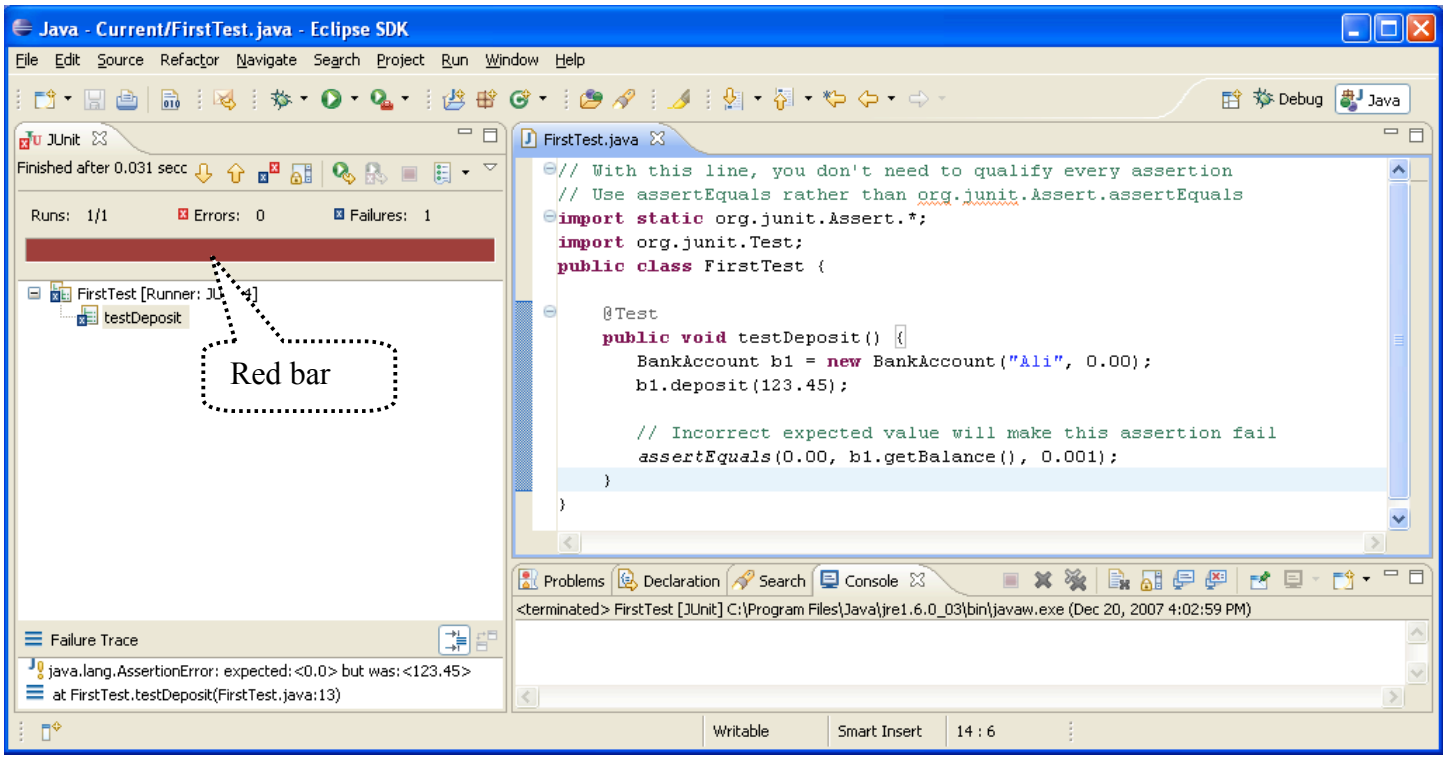
An assertion passes when the actual value equals the expected value in `assertEquals`.

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

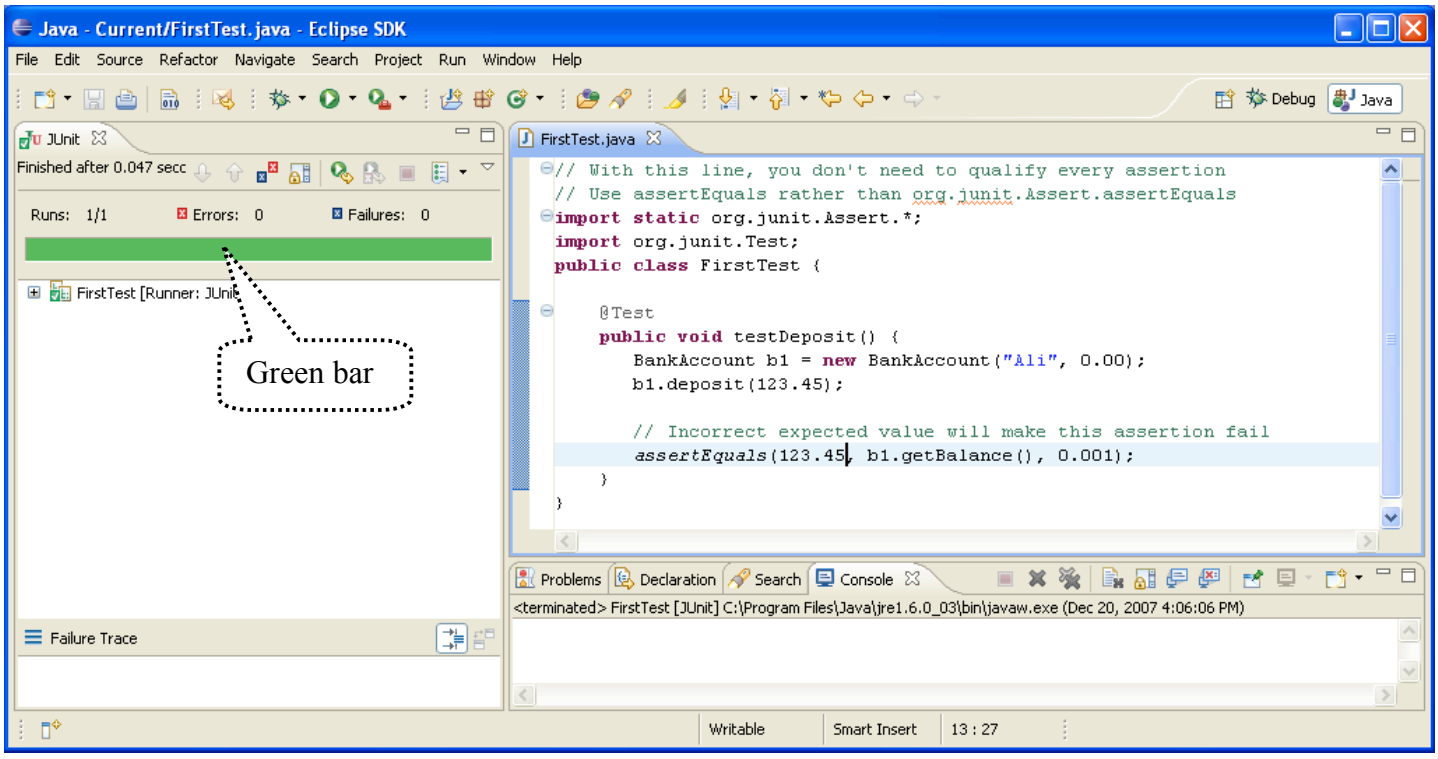
An assertion failes when the actual values does not equal the expected value.

```
  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 "the"` 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());
  }

  // . . . more test methods will appear below
}
```

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.

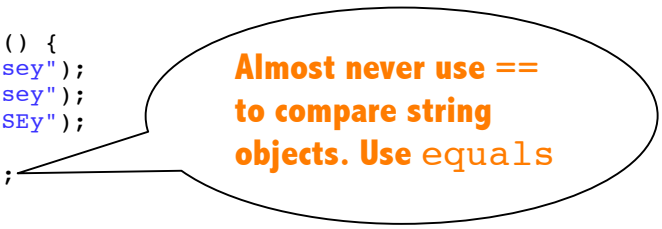
Use an assignment if you want to change the String reference to refer to a different String.

```
@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`

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));
}
```



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 references, not the values of the objects. The above code generates two different String objects that just happen to have the same state. Use the `equals` method of `String`. The equals method was designed to compare the actual values of the string—the characters, not the reference values.

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

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
System.out.println(b1.getBalance);                 // i
```

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

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

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.
-i Missing `()`.

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)

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.

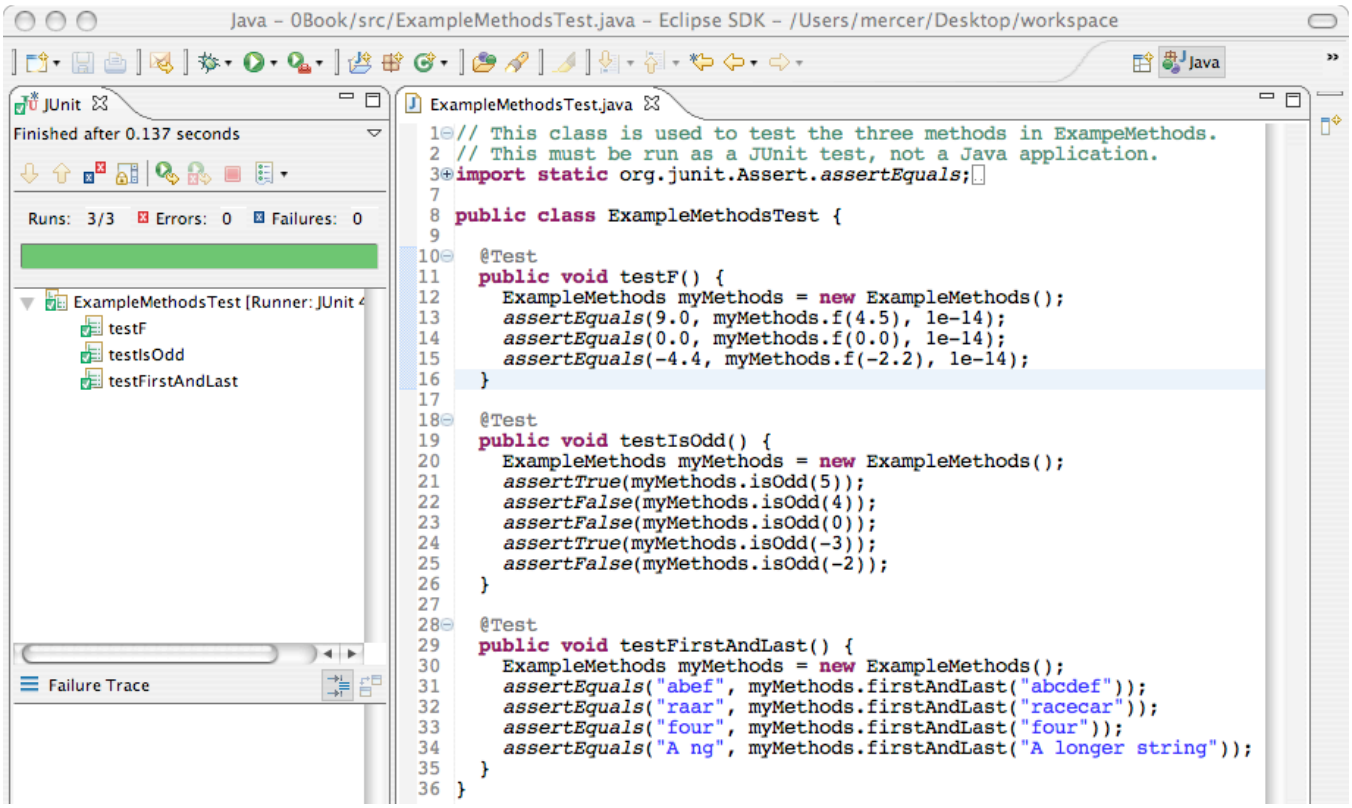
As suggested in Chapter 1, testing can occur at many times during software development. When you write a method, test it. For example, a test method for `firstAndLast` could look like this.

```
  @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 is a relatively new way to implement and test methods 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

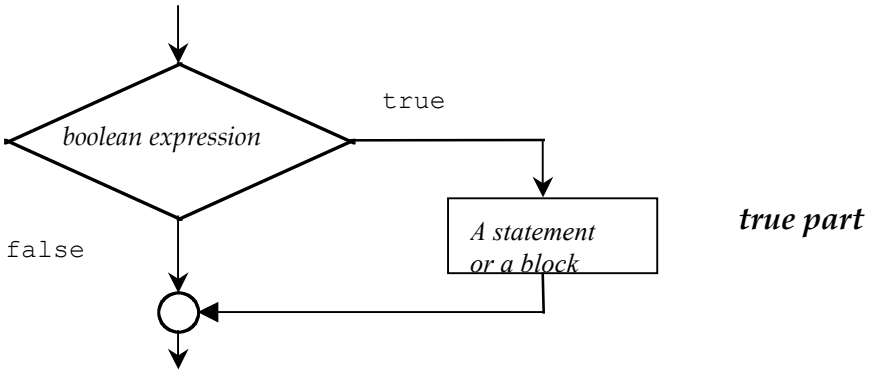
A *Boolean-expression* is any expression that evaluates to either true or false. The *true-part* may be any valid Java statement, including a block. A block is a sequence of statements within the braces `{` and `}`.

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

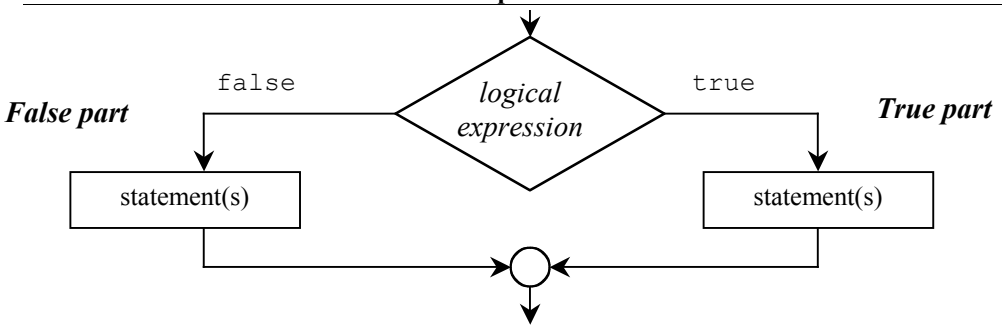
The `if else` statement is an `if` statement followed by the alternate path after an `else`. The *true-part* and the *false-part* may be any valid Java statements or blocks (statements and variable declarations between the curly braces `{` and `}`).

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:

```
  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
```

These two methods do branch and boundary testing.

```
@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

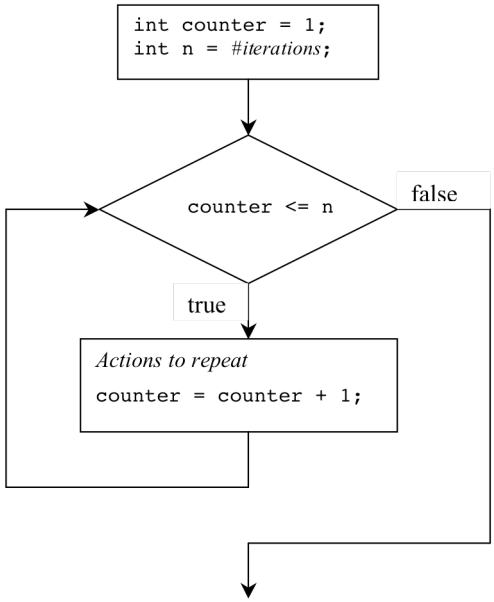
```
int start = 1;
int end = 6;
while (start < end) {
  System.out.println(start + " " + end);
  start = start + 1;
  end = end - 1;
}
```

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.
```

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.

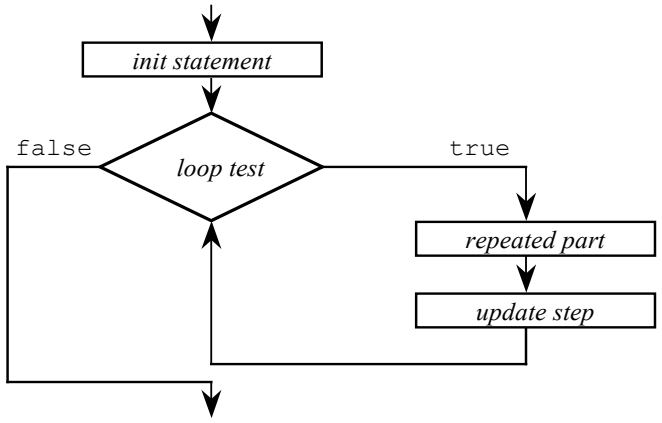
```
// Predetermined number of iterations
int n = 5;
for (int j = 1; j <= n; j = j + 1) {
  // Execute this block n times
}
```

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

4. declare and use arrays that can store reference or primitive values
5. 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
```

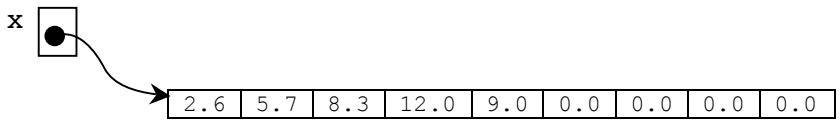
Each array element is a variable of the type declared. Therefore, these two integers will be promoted to double before assignment.

```
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

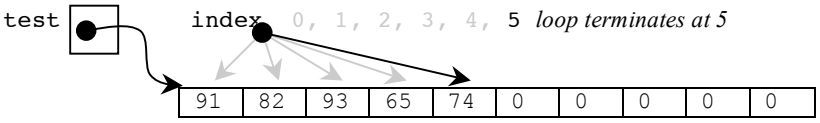
Programmers must frequently access consecutive array elements. For example, you might want to display all of the meaningful elements of an array containing test scores. The Java `for` loop provides a convenient way to do this.

```
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.

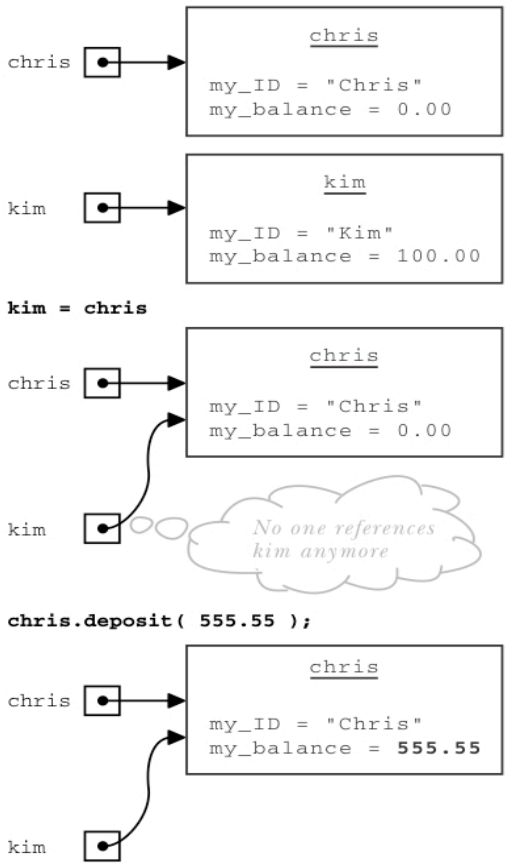
The following code mimics the same assignments that were made to the primitive variables above. The big difference is that the `deposit` message sent to `chris` actually modifies `kim`. This happens because both reference variables `chris` and `kim`—refer to the same object in memory after the assignment `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.

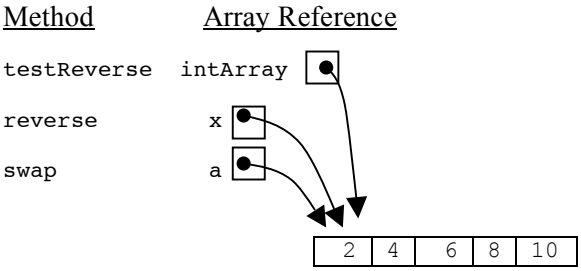
```
@Test
public void testAddToBalance() {
  BankAccount kim = new BankAccount("Chris", 0.00);
  assertEquals(0.0, kim.getBalance(), 0.0001);
  increment(kim);
  assertEquals(555.55, kim.getBalance(), 1e-14);
}

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"   Because "LISA" is greater than name[mid],
name[1]    "CLAY"  the objects name[0] through name[3] no longer
name[2]    "KIM"   need to be searched through and can be eliminated from
name[3]    "LAU"   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

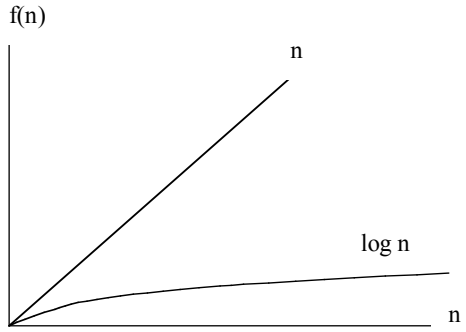
n would require potentially twice as many loop iterations for sequential search. However, the same doubling of n would require potentially only one more comparison for binary search.

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

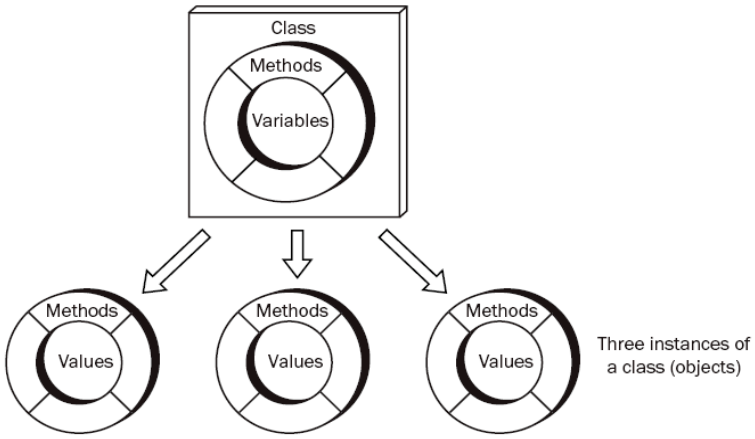
- Implement Java Classes as a set of methods and variables
- Experience designing and testing a class that is part of a large system

---

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.

```
// 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?

```
@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

Instance Variables	State of `bagOfStrings`
data[0]	"A string"
data[1]	"Another string"
data[2]	"and still another"
data[3]	null // next available
data[4]	null
...	...
data[9]	null
n	3

After

Instance Variable	State of `bagOfStrings`
data[0]	"A string"
data[1]	"Another string"
data[2]	"and still another"
data[3]	**"and a fourth"**
data[4]	null // next available
...	...
data[9]	null
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.

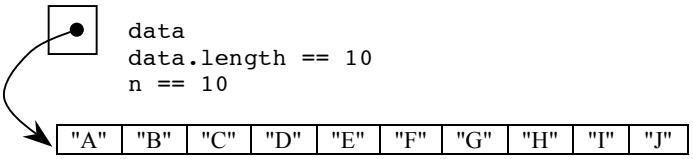
`private void growArray()`

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:

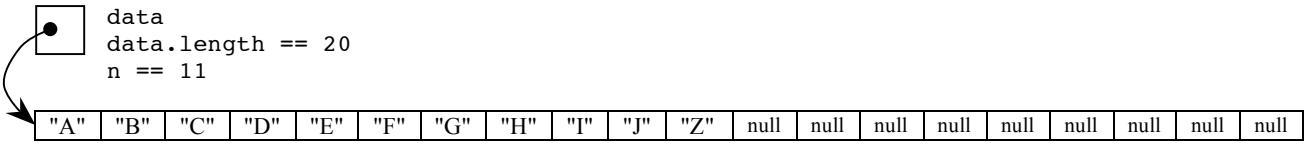
- Make a temporary array that is bigger (by 10) than the instance variable.
- Copy the original contents (`data[0]` through `data[n - 1]`) into this temporary array.
- Assign the reference to the temporary array to the array instance variable

```
// 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" .. "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 consruction (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 athis 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` such that stores 120 doubles 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` that will print the elements in 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 text file "quiz3by4" has these 4 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;
}
```