# Essentials of the Java Programming Language

Joan Krone
Thomas Bressoud
R. Matthew Kretchmar
Department of Mathematics and Computer Science
Denison University

# Chapter 1

# Introduction

## 1.1 Preliminaries

### 1.1.1 Learning a Language

Programming a computer is both a creative activity and a process structured by rules. Computers are *programmed* or given instruction, through the use of programming languages, so to program a computer, one must learn a programming language. The goal of this book is to introduce a programming language called Java.

Learning a programming language has similarities to learning a natural language, such as English or Spanish or Japanese. Natural languages have a *lexicon*, a *syntax*, and a *semantics*. The lexicon is the vocabulary and punctuation. The rules of syntax dictate how the lexicon can be ordered to form correct sentences. The semantics conveys the meaning when the chosen words are combined in the chosen syntax of each sentence. The semantics changes, sometimes in subtle ways, depending on both the words chosen and the sentence syntax.

So too, a programming language has lexical elements, a syntax, and a semantics. The lexical elements are comprised of keywords in the programming language, symbols such as arithmetic operators or parenthesis or braces, and words denoting identifiers by which we can name what is being operated upon. The syntax specifies precise rules for how the lexical elements are ordered to form correct language statements and programs. The semantics is the meaning of the constructed language statements. In this context, the meaning is operational, telling the computer "what to do" as the statements are encountered, in a well defined sequence, in the execution of the program.

Learning programming languages and learning natural languages share other similarities. First, the typical method for learning a language is a spiral approach.

We start with a very limited vocabulary, a few simple syntactic constructs and their semantics. We then build our repertoire, adding vocabulary, syntax, and semantics, and often returning to previously covered topics. Second, mastering a language takes practice through active participation. You should practice by repeating each example and by working out the exercises and problems posed in the book. Finally, *memorization* of example patterns is no substitute for understanding the semantics associated with each syntactic structure. As the syntax changes, we must understand the corresponding change in the semantics.

### 1.1.2 The Program and the Computer

Learning a programming language, and particularly learning your *first* programming language, is intimately tied with learning how a computer works. Understanding how a computer works involves understanding how numbers and characters and aggregate data may be encoded and stored in the memory of the computer, and how the data stored in memory can be operated upon by the central processing unit (CPU) executing millions of very simple instructions.

Programming languages such as Java, C, or C++ are known as *high-level* languages. They allow specification of program instructions in a manner closer to natural languages, but without the ambiguity and lack of precision. Computers, on the other hand, are only capable of executing exceedingly primitive instructions in a *low-level* language known as the computer's *machine language*. We bridge the gap by translating a high-level language into a low-level language through a program known as a *compiler*.

## 1.2 Problem Solving through Programming

Programming is about *solving problems*. Regardless of the language used, this activity proceeds through a set of well-defined stages during the development of the program.

**Design** In the design stage, a careful problem statement is articulated, indicating the desired outcomes of the program. The problem statement is then refined by listing a set of steps designed to accomplish the goal. The set of steps is called an *algorithm*, and may be presented in English or in some other appropriate notation. The design stage does not require the use of a computer.

**Edit** In the edit stage, a program is created in a text editor and stored in one or more files on disk. These files are known as the program *source code*.

**Compile** During the compile stage, the compiler translates the program source code in the source file(s) into the machine code appropriate for the machine on which the program will execute.

**Execute** The execute stage is the time when the compiled program is loaded from disk into its execution environment and the machine carries out the instructions.

By its very nature as a human endeavor, problem solving is rarely completely successful on the first attempt. Some or all of the above phases may be revisited as we refine and iterate toward a correct solution.

The above development description applies to any programming language, not just Java. It is worthwhile to make some additional observations about program development specific to Java.

- The source code for a Java program are contained in text files with the `.java` extension.

- The Edit, Compile, and Execute stages of program development can sometimes be brought together in the context of a computer program known as an *Integrated Development Environment* (IDE). Such IDEs vary in power and facilities provided to the developer, and many are overkill for a student's first introduction to a programming language. There are two Java IDEs that are well suited to a new student, **Dr. Java**[1] and **Blue J**[2].

- Many common programming subproblems have already been solved and it is not necessary to build these again. For Java, an extensive set of libraries is available and is part of the what is known as the *Java Platform*. The *Java Application Programming Interface* (API) defines the facilities available for use from our Java programs.

- Different hardware platforms, such as Intel x86, Mac G4/5, or SGI MIPS, have different machine languages. For Java to be able to run without modification on multiple platforms, the Java platform includes a *Java Virtual Machine* (JVM). This is a computer program that simulates a machine that is specific to Java. The machine code for the JVM is called *bytecode* and when we compile a Java program, the compiler generates the bytecode and places it in a file with a `.class` extension.

- During the Execute stage, the class file for our program is loaded into the JVM, and combined with the class files that realize the Java API.

The remainder of this chapter is a short tutorial, introducing you to a small set of elements of the Java language. The intent is to give you an initial flavor of programming in Java without getting immersed in too many details and formal rules. Think of it as your very first increment of vocabulary, syntax, and semantics of the language. The goal is to quickly get to the point where you can write simple, but complete, programs. Subsequent chapters of the book will delve into all of the elements touched upon here, and in much greater detail.

## 1.3 A Java Tutorial

This tutorial will lead you through a sequence of Java programs designed to introduce some basic concepts of Java programming. We begin with an application to print a welcome message.

### 1.3.1 Welcome Application

#### Design

Your first application, `WelcomeApp`, has the simple problem statement:

> Print the string of characters "Welcome to Java Programming!" to a text console window.

As we shall see, part of the Java system API includes a Java statement that allows us to print strings of characters as output to the text console window. Thus we can complete our design stage with a single step:

1. Print the string `"Welcome to Java Programming!"`.

#### Edit

The following Java language code accomplishes the goal of our first program. The line numbers on the left are included only for our reference; they are not part of the source program itself.

---

```
1  /**
2   * The WelcomeApp class implements a simple application
3   * that prints a welcome message to the standard output.
4   */
5  public class WelcomeApp {
6
```

```
 7    // The WelcomeApp class consists of a single method,
 8    // namely the application entry method called 'main'.
 9
10    public static void main(String[] args) {
11
12      // Invoke a system-provided method to
13      // print a String argument to standard output.
14
15      System.out.println("Welcome to Java Programming!");
16    }
17  }
```

---

Successful completion of the edit stage requires the creation of the source code text file whose contents are the Java code given above. The file must end in the `.java` extension and, due to requirements discussed later, *must* be named `WelcomeApp.java`.

DR. JAVA

The edit, compile, and execute stages may all be accomplished within the Dr. Java Integrated Development Environment. First, you must launch the Dr. Java application. Depending on how Dr. Java was installed, there may exist a shortcut or desktop icon that may be used to launch the application.

Once the application is launched, you will see three primary window panes in the Dr. Java window. On the left side is the *Files Pane*, a pane that will display the set of Java source files that are currently active in Dr. Java. Upon initial launch, there are no active source files, and `(Untitled)` should appear in this window pane indicating an empty unnamed Java source file is ready for entry of code. The largest window pane, on the right, is the *Definitions Pane* and displays the content of a single active source file. Since we have not entered any Java code yet, this pane will be empty. The window pane across the bottom of the application window starts in the tab for Dr. Java Interactions, and is referred to as the *Interactions Pane*.

Click in the Definitions Pane and enter the given Java code exactly as it appears above (but without the listing line numbers). When you make modifications to a source file, the condition of having an unsaved source file is indicated by a '`*`' appearing next to the source file in the File Pane. As you type code in the Definitions Pane, you may note that parts of the code will be displayed in different colors. This is called *syntax highlighting* and is used to show different parts of the vocabulary of the Java language and their place in the syntax of the source file.

Once the Java code has been entered, the source file must be saved. This operation writes the text source file out to the file system on the disk of the computer. From the `File` menu, select the `Save` menu item. You can also click the Save button on the button bar that appears underneath the Menu bar. This will result in a Save dialog box. The 'File Name' and the 'Files of Type' items will be filled in with "WelcomeApp" and "Java source files". These are the correct entries and should not be changed. This is specifying that the name of the file is `WelcomeApp.java`. Use the upper portion of the Save dialog box to navigate to the folder/directory where you want the source file (and ultimately the bytecode file) to reside. Then click the `Save` button. Upon successful completion of the save, the File Pane will change to `WelcomeApp.java`.

---

**Compile**

If you have successfully entered and saved the Java source code *exactly* as it appears, then the compile stage should be very straightforward. We only have to specify the source file to the compiler, which will, without any further interaction, generate the output bytecode file. Problems arise in the form of compiler errors if typographical errors have resulted when the code was entered. If this occurs, compare your code carefully with the given code; correct any errors, and retry the compile step given below.

DR. JAVA

Ensure that WelcomeApp.java appears and is highlighted in the Files Pane and that the Java source code that you entered appears in the Definitions Pane of the Dr. Java application. This should be the current state of affairs following the Edit stage. If Dr. Java had been closed after the Edit stage, you can return to this point by selecting the `Open` menu item from the `File` menu and navigating to, and opening, the WelcomeApp.java source file from the prior step.

Compile the source file by clicking the `Tools` menu and selecting either the `Compile All Documents` or the `Compile Current Document` menu item[1]. During compilation, the bottom pane will switch to the "Compiler Output" tab and display the message "Compilation in Progress, please wait ...". If there are no syntax errors, the completion of the compilation will be indicated by the

[1]These two are equivalent when there is a single file listed in the Files Pane. They differ when more than one source file is active, and thus the Files Pane contains more than one entry.

message “Last compilation completed successfully” in the Compiler Output tab of the bottom pane.

---

**Execute**

Now for the moment of truth. It is time to execute your first Java program.

DR. JAVA

Again, ensure that the `WelcomeApp.java` appears and is highlighted in the Files Pane and that the Java source code that you entered appears in the Definitions Pane of the Dr. Java application. Now, from the `Tools` menu, select the `Run Document's Main Method` menu item. The bottom pane should change to the “Interactions” tab, and you will see, at the ’>’ prompt of the Interactions pane, the following command:

```
java WelcomeApp
```

This should be followed by the output, `"Welcome to Java Programming!"` in your console window. This is the result of the successful execution of your Java program. Alternatively, you can click on the “Interactions” tab of the bottom pane and manually type the `java WelcomeApp` command. When you press Enter/Return, the program will execute and display the `"Welcome to Java Programming!"` output.

---

**A Closer Look at the WelcomeApp Java Code**

Now for some explanation of the source code. This first program exhibits four elements of the language: comments, class definition, method definition, and method invocation. All four of these elements will be common to almost every Java program.

A *comment* is free-form text that adds description to a program suitable for helping a reader to understand what parts of the program are designed to do. Lines 1-4 together form a syntactic unit that is delimited at the beginning by the character sequence ‘`/**`’ and at the end by the character sequence ‘`*/`’. This defines a multi-line comment in Java. Between the beginning of comment delimiter and the end-of-comment delimiter, the programmer may type any sequence of characters available from the keyboard. Lines 7, 8, 12, and 13 are also Java comments,

but these are single-line comments. The comment begins with the the character sequence '`//`' and continues until the end of the current line. In this example, the single line comments are the only element on the line, but this style of comment is often used following some non-comment Java syntax and is used to explain the Java on the line on which it resides.

Lines 6, 9, 11, and 13 are blank lines. The use of blank lines and the use of indentation to show nesting of syntactic structures improves the readability of the source code, but has no impact on the instructions that will be executed. Neatness and readability count when we are writing programs.

The ability to cope with problems of any significant size requires the ability to organize and break a problem up into smaller, more manageable, subproblems. All high-level languages have some facilities for accomplishing this, and in Java, the fundamental facilities are the notions of a class and of a method. A *method* gathers together a set of program steps that define an action in the program. A *class* is a collection of methods.

Line 5 begins the definition of the class named `WelcomeApp`. In that line `public` and `class` are necessary Java keywords, `WelcomeApp` is providing the name of the class, and the '{' is known as a *delimeter* that begins the contents of the class. The '}' on line 17 is the corresponding closing delimeter ending the class definition.

Line 10 begins the definition of the method named `main`. The keywords `public`, `static`, and `void` are required syntax in the definition of this method, as is the syntax '`(String[] args)`' following `main`. Like defining a class, the definition of a method delimits the beginning and end of its contents with the symbols '{' and '}'.

Every Java program must consist of at least one class, and there must exist a method named `main` declared as we see on line 10. Chapters 8 and 9 will treat the concepts of class and method in much greater detail.

Now that we have covered all the syntax providing the required structure of a class definition for `WelcomeApp` and its `main` method definition, we are left with a single statement, on line 15, whose execution actually accomplishes our goal.

```
System.out.println("Welcome to Java Programming!");
```

This Java statement is the invocation of the method `System.out.println()`. (Or, more simply, `println()`). A method invocation has the semantics of temporarily suspending execution of the steps in the current method and executing the steps defined in the invoked method. Once those steps are complete, execution continues in the current method at the point following the method invocation statement.

In this example, the string `"Welcome to Java Programming!"` is the parameter that is passed during method invocation to the `println()` method. The println method is defined as having a string parameter and the method handles the output operation of printing the given string to the output console. The method is also defined to end the output line and start the next line, so that subsequent output is on a new line. By specifying different strings as the passed parameter to the `println()` method, one can have any such string printed to the output console.

**Exercise 1.1** *Create (i.e. Edit, Compile, and Execute) a new Java application class, called* `Welcome2` *whose goal is to print the following output on the console:*

> *Welcome*
> *to Java Programming!*

**Exercise 1.2** *Create a new Java application, called* `Bill` *whose goal is to print the following output on the console:*

> *All the world's a stage,*
> *And all the men and women merely players:*
> *They have their exits and their entrances;*
> *And one man in his time plays many parts,*
> *His acts being seven ages.*

**Exercise 1.3** *Let us intentionally make some mistakes. One at a time, make the following changes to the original (correct) source file,* `WelcomeApp.java` *and then attempt to compile the modified program. Observe the errors that result:*

1. *Omit the semicolon on the end of the line that begins* `System.out.println` *(line 15).*
2. *Omit the word* `void` *on line 10 before the word* `main`*.*
3. *Omit the symbol sequence '*/' on line 4.*
4. *Add the word* `new` *between the words* `public` *and* `class` *on line 5.*

Another method, named `System.out.print()` (`print()`) is available and is a variant of the `println` method we have already seen. Like its cousin, it also takes a given string parameter and prints the string to the output console. However, it does not add the newline to begin the next line of the console, but leaves the current output point immediately following the printed string, so any subsequent output will follow on the same line.

Using the `print()` method, we can use multiple method invocations to achieve the same effect as a single (longer) invocation of `println()`. So, the statement sequence

```
System.out.print("Welcome to ");
System.out.println("Java Programming!");
```

is equivalent to

```
System.out.println("Welcome to Java Programming!");
```

**Exercise 1.4** *Modify your first* `WelcomeApp` *class so that the body of the* `main` *method uses the two statement sequence of a* `print()` *followed by a* `println()`.

**Exercise 1.5** *Modify your* `WelcomeApp` *class once again so that the* `"Welcome to "` *string in the* `print()` *invocation leaves off the trailing space. What do you expect to happen? Compile and Execute the application and see if you are correct.*

### 1.3.2 A Conversion Application

#### Design

The second application of this tutorial has slightly more depth than the simple printing of output. We are going to use the computer to perform some calculations in our program as well.

Consider the problem of converting distances. Some of the most common distances used in road races include 5K, 10K, and occasionally a 5 mile race. We wish to convert the kilometer distances to miles and the mileage distances to kilometers. This will be the goal of our next application, with the problem statement:

> Compute and print the conversion of 5K, and 10K to their corresponding mileage distances, and 5 miles to its corresponding kilometer distance.

This problem statement can be refined into the following steps.

1. $x$ miles = 5K * 0.6214 miles/K
2. Print the string `"5K = "` and then $x$ and then the string `" miles"`.
3. $y$ miles = 10K * 0.6214 miles/K
4. Print the string `"10K = "` and then $y$ and then the string `" miles"`.
5. $z$ K = 5 miles * 1.609 K/mile

6. Print the string `"5 miles = "` and then $z$ and then the string `" K"`.

Note the introduction of the variables $x$, $y$, and $z$ during the enumeration of steps so that we have a way of naming and talking about a computed quantity to use in subsequent steps. At this point, these variables have nothing to do with a computer program. They are simply tools used in the same way that we use variables in algebra.

**Edit**

The following Java language code accomplishes the goal of the second program of our tutorial.

---

```
1  /**
2   * The ConvertMilesK class implements a simple application whose
3   * purpose is to convert the distances of 5Km and 10Km to miles, and
4   * to convert 5 miles to Km.
5   */
6  public class ConvertMilesK {
7
8    // The ConvertMilesK class consists of a single method definition,
9    // namely the application entry method called 'main'.
10
11   public static void main(String[] args) {
12
13     double milesValue;      // variable for miles in conversion
14     double kilometersValue; // variable for kilometers in conversion
15
16     // Convert 5Km to miles and print result
17
18     kilometersValue = 5.0;
19     milesValue = kilometersValue * 0.6214;
20     System.out.println("5Km = " + milesValue + " miles");
21
22     // Convert 10Km to miles and print result
23
24     kilometersValue = 10.0;
25     milesValue = kilometersValue * 0.6214;
26     System.out.println("10Km = " + milesValue + " miles");
27
28     // Convert 5 miles to Km and print result
29
30     milesValue = 5.0;
31     kilometersValue = milesValue * 1.609;
32     System.out.println("5 miles = " + kilometersValue + " Km");
```

```
33     }
34   }
```

---

Using the skills acquired during the last application of the tutorial, create the `ConvertMilesK.java` source file based on the Java code given above as appropriate to your development environment.

### Compile and Execute

Compile the created `ConvertMilesK.java` and then execute. You should obtain output similar to the following:

```
5Km = 3.1069999999999998 miles
10Km = 6.2139999999999995 miles
5 miles = 8.045 Km
```

### A Closer Look at the ConvertMilesK Java Code

Note the similarities between this program and the earlier `WelcomeApp` program.

- Both source files begin with a comment that describes the purpose of the class, and is often a restatement of the problem statement.
- Line 6 of `ConvertMilesK.java` beginning the class definition is almost identical to Line 5 of `WelcomeApp.java`. The difference is simply in the name of the class being defined.
- The beginning of the `main` method definition on line 11 is identical to the corresponding line 10 of `WelcomeApp`.

As we examine the Java instructions within `main` method (lines 12 through 32) in the `ConvertMilesK` class, we can readily observe a correspondence between the Java code and the steps enumerated during the design stage. In particular, lines 16 through 32 correspond to the conversion and print for the 5Km, 10Km, and 5 mile conversions.

We can create variables in a programming language that we can use for naming and manipulating values during the steps of the program. In this program, we define and use two variables, named `milesValue` and `kilometersValue`.

Lines 13 and 14 are *variable declaration statements* (also called simply *declarations*) for variables `milesValue` and `kilometersValue`. A variable declaration statement associates a programmer supplied variable name, or *identifier*, with a unit of storage in the memory of the machine.

A declaration also associates a data type with the declared variable. In programming languages, the *type* of a variable determines the set of valid values that are permitted to be stored in the variable. The type of both `milesValue` and `kilometersValue` is given by the keyword `double`. The type `double` for a variable specifies that the variable may have real number values, where real numbers are numbers that contain decimal points (e.g. 1.5, -8.23, 0.0).

In general, a variable may be declared using the following syntax:

```
<type> <identifier>;
```

substituting the appropriate data type for <*type*>, and the desired variable name for <*identifier*>.

In line 18, we assign the constant real number value 5.0 to the variable that was declared in line 14, `kilometersValue`. This is our first example of an *assignment statement*. The assignment statement is a fundamental construct of programming. It updates the storage unit associated with a variable with a value.

The syntax of an assignment statement is the following:

```
<identifier> = <expression>;
```

The symbol '`=`' is the *assignment operator*. For the expression side of an assignment statement, we can have expressions that are as simple as a constant value, but can be considerably more complex. Arithmetic and more complex expressions will be explored in Chapter 3.

The semantics of the assignment statement is the following:

1. Evaluate the right hand side of the assignment. Use the current values associated with any variables that appear and combine them with the operators and constants and obtain a final value and type for the complete expression.

2. Store the computed value at the location associated with the variable named on the left hand side of the assignment.

Line 19 demonstrates another assignment statement. By the semantics given above, the current value of `kilometersValue`, which is 5.0 because of the previous statement, is multiplied by the constant value 0.6214. The result is stored in the variable `milesValue`.

Then, on line 20, we see a statement invoking the `println()` method. The difference from what we have seen before is that the string argument to the method is specified in pieces. We have the constant string values `"5K = "` and `" miles"`. When working between operands that are strings, the '`+`' operator performs string concatenation, which appends two string together into a composite string. The use

of the variable `milesValue`, whose type is `double`, in a string expression like this causes the current value of `milesValue` to be converted to a string, so that the final result is a string argument that can be passed to `println`.

The remaining lines of the program, lines 22 through 32, repeat the same kinds of assignments and print statements. Note that it is perfectly fine to reuse variables.

**Exercise 1.6** *Study the remainder of the ConvertMilesK* `main` *method definition, looking for the similarities and differences between these remaining sections and the section discussed.*

**Exercise 1.7** *Modify the ConvertMilesK program to perform conversions for 15K, 25K, 10 miles, and the official marathon distance of 26 miles, 385 yards.*

**Exercise 1.8** *Create a new class called* `ConvertFtoC` *to convert between temperatures in Fahrenheit and their corresponding temperatures in Celsius. Convert the values 80F, 72F, 32F, and 0F. Be sure and use appropriate names for your variables.*

# Chapter 2

# User Interaction

## 2.1 Console Based User Interaction

### 2.1.1 An Application to Add Two Integers Input by the User

#### Design

The next example application addresses the need to interact with a user, getting their input in order to achieve a goal. Say that we want to add two numbers. However, we don't know ahead of time what those numbers will be. We need some mechanism in our program to ask the user for some input, and then to retrieve the user's response. We will use the `print()` method to print the question asking for input to the user, but we need a corresponding input method to retrieve the response.

Our problem statement:

> Add two integer numbers input by the user, and print the computed sum.

can be realized by the following steps:

1. Prompt the user for the first integer.
2. Retrieve the first integer, denoted $x$.
3. Prompt the user for the second integer.
4. Retrieve the second integer, denoted $y$.
5. Compute $z = x + y$
6. Print the string `"Sum is "` followed by the value of $z$.

**Edit**

*Java (J2SE) 5.0*

When programming in Java 1.5.0 (J2SE 5.0) and later, the following Java program satisfies the requirements of this application.

```
1  /**
2   * The AddTwoInts class implements a simple application
3   * whose purpose is to input two numbers entered by the
4   * user, compute the sum, and print it out to the console.
5   */
6
7  import java.util.Scanner;  // program needs the Scanner
8                             // class for console input
9  public class AddTwoInts {
10
11   // The single method of the AddTwoInts class, main.
12
13   public static void main(String[] args) {
14
15     int firstNumber;
16     int secondNumber;
17     int sum;
18
19     // In J2SE 5.0, can use the Scanner class
20
21     Scanner consoleIn;
22     consoleIn = new Scanner( System.in );
23
24     System.out.print("Enter first integer: ");
25     firstNumber = consoleIn.nextInt();
26
27     System.out.print("Enter second integer: ");
28     secondNumber = consoleIn.nextInt();
29
30     sum = firstNumber + secondNumber;
31     System.out.println("Sum is " + sum);
32   }
33 }
```

Create the `AddTwoInts.java` source file based on the given code as appropriate to your development environment.

### Compile and Execute

Compile the created `AddTwoInts.java` and then execute. You should obtain output similar to the following:

```
Enter first integer:  34
Enter second integer:  -12
Sum is 22
```

where the boldface values of 34 and -12 are the input typed by the user.

### A Closer Look at the AddTwoInts Java Code

The `main` method of `AddTwoInts` begins with three variable declarations, for the variables `firstNumber`, `secondNumber`, and `sum` (on lines 13–15). These associate the given variable names with a unit of storage and with a data type of `int`. The `int` data type is used for whole numbers, which have no fractional part. Compare these to the declarations of the `double` variables in the example in Section 1.3.2.

*Java (J2SE) 5.0*

The `main` method continues with another variable declaration and an assignment statement. The variable declaration uses a data type of `Scanner` and a variable name of `consoleIn`. The behavior of a `Scanner` as a data type is given by the `Scanner` class of the Java API. Using classes to define types allows more complex data structures and methods to be created, and in this case, defines the operations needed to get user input. Just as with the primitive types of `double` and `int`, this `Scanner` variable is uninitialized, and its initialization is the purpose of the following statement. The expression on the rhs of the assignment creates a new `Scanner` object based on the console input stream `System.in`. Once the object is created, we have access to the methods defined in the `Scanner` class through that object.

---

Line 17 outputs a string to the console. Note the use of `print()` instead of `println()` so that the following user input is on the same line as the prompt.

The next line is an assignment statement in which the right hand side is an expression made up of a method invocation. Unlike the method invocations we have seen up to this point, the method must return a value and have an associated data type in order for the assignment statement to follow its prescribed semantics.

*Java (J2SE) 5.0*

The method `consoleIn.nextInt()` is defined to retrieve input typed by the user, convert the user's keystroke sequence up to the next Enter into an integer, and return the integer as the value computed by the method. When we invoke a method, we pass any paramters needed by the method inside the parentheses following the method name. In this case, there are no arguments, and so we simply have '`()`'.

---

The next two lines, lines 20 and 21, of the Java program repeat the process of prompting the user for an integer and then retrieving the input from the user through a method invocation and assignment into the variable `secondNumber`.

Once the two values to be added are stored in the variables `firstNumber` and `secondNumber`, we are ready to compute their sum. This is accomplished on line 23 with an assignment statement whose target is the variable `sum` and whose computation is given by the expression, evaluating the current values of `firstNumber` and `secondNumber`, and adding them together to get a value.

In similar fashion to earlier examples, the value of the variable `sum` is then printed out to the console.

**Exercise 2.1** *We have seen the use of integer variables and the corresponding methods to retrieve an integer (type `int`) value as input from the user. We may also wish to retrieve a `double` value as input from the user.*

*Java (J2SE) 5.0*

*The `Scanner` class includes a method called `nextDouble()` for input of a `double` value from an object that has type `Scanner`, such as `consoleIn` from the previous example. It is used in exactly the same manner as `nextInt()`.*

---

*Create a new application class called `AddTwoDoubles` that declares the variables `firstNumber` and `secondNumber` and `sum` to be of type `double` and*

*modify the method invocations to retrieve the user input to call these* `double`*-based methods. Neither the addition operation nor the* `println` *of the result needs to change.*

**Exercise 2.2** *Create a new application program to convert a single value from Celsius to Fahrenheit. The type of the value should be a* `double` *and it should be input by the user. Use appropriate variable names and a prompt that informs the user of what is being asked for.*

## 2.2 Dialog Based User Interaction

For current students, who have grown up with computers and have been using computers for word processing, web browsing, building spreadsheets, and so forth, the console-based input and output of Section 2.1 may well seem quite foreign and even primitive. Users of today are much more accustomed to Graphical User Interfaces (GUIs), in which computer programs use windowing systems and a collection of graphical interfaces such as menus, menu items, toolbars, and dialog boxes in which to display information to the user and to gather information from the user.

Graphical User Interfaces use a model of programming, called *event-driven programming*, that is at level of sophistication beyond the initial abilities of a student learning their first programming language. This is why we begin by demonstating interaction with the user through input and output of a text console.

However, many students may desire a more graphical approach. The remainder of this chapter introduces the use of a simple but relatively powerful collection of system-provided APIs for dialog boxes. These can be used without forcing the programmer into an event-driven model, and can be used to display messages to the user, as well as gather information from the user, sufficient for the types of interaction required through the rest of this book.

### 2.2.1 Basic Operations

The Java API provides a class called `JOptionPane`, which allows user interaction through dialog boxes. To allow a program to use these facilities, the program must include the statement

```
import javax.swing.JOptionPane;
```

before the class definition begins.


Figure 2.1: JOptionPane Message Dialog

### Displaying a Message to the User

A dialog box displaying a message to the user is comparable to our use of `System.out.println`. We wish to construct a message string and have a dialog box displayed with the message, and an 'OK' button, by which the user may dismiss the message.

The method `JOptionPane.showMessageDialog()` provides this facility. The method takes two parameters, but the first is only used in a full GUI, so we use the keyword `null` to indicate our standalone use. The second parameter is a string specifying the message.

The following statement

```
JOptionPane.showMessageDialog(null, "Welcome to Java");
```

results in the display of the dialog box shown in Figure 2.1.

### User Confirmation

Another common interaction with a user is to ask a question, expecting a response of "yes" or "no" or "cancel". The method `JOptionPane.showConfirmDialog()` provides this facility. Like *showMessageDialog()* the method takes two parameters, and we use a value of`null` to indicate our standalone use. The second parameter is a string specifying the question.

The method interprets the button push of the user and maps a "Yes" button press to the integer value 0, a "No" button press to the integer value 1, and a "Cancel" button press to the integer value 2. It is this integer value that is "computed" and returned as a result when the method is invoked.

The following code declares a variable name `userValue` of type `int` and then invokes the `showConfirmDialog()` method, assigning the result to the `userValue` variable. The resulting dialog box is shown in Figure 2.2.

```
int userValue;
userValue = JOptionPane.showConfirmDialog(null, "Do you wish
to continue?");
```

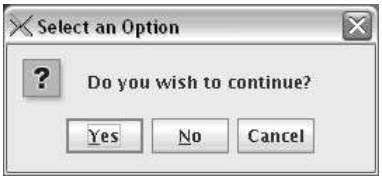



Figure 2.2: JOptionPane Confirm Dialog

If the user pressed the "No" button, the `userValue` variable would have the value 1 as a result.

### User Input

General input can be retrieved from the user by displaying a dialog box that has a text entry area. `JOptionPane.showInputDialog()` is a method that provides this general input. In its simplest form, it also has two parameters, a `null` for its standalone (non-GUI) use, and a string specifying a prompt displayed to the user. Since `showInputDialog()` is a general facility, it simply collects the string of characters typed by the user in the text entry area and returns that string as the result of the method invocation.

Just as we have strings of characters that we enclose in double quotes to use in our programs, we can have variables whose type is `String` and that we can use to manipulate strings. In the following code, we declare a variable, `userAnswer`, to have a data type of `String` and then use the `showInputDialog` to retrieve the string of characters from the user and assign it to the declared variable.

```
String userAnswer;
userAnswer = JOptionPane.showInputDialog(null,
"Enter your name:");
```

Figure 2.3 shows the result of this invocation and after a user has typed in the characters "Susan" into the text entry area. Note that the user does not type the double quote marks.

While the demonstrated usage is adequate for user input of strings, what do we do when we require other data types, like an integer (`int`) or a real number (`double`)? The answer is that we use the same method and retrieve a string from the user, but we then need to convert the string into the required data type.

Suppose we want to use dialog-based input for the application of Section2.1.1. For each integer, we want to prompt the user for the integer and then store the number into a variable. We begin as we did in the last example:


Figure 2.3: JOptionPane Input Dialog (of a string)


Figure 2.4: JOptionPane Input Dialog (of an integer)

```
String userAnswer;
userAnswer = JOptionPane.showInputDialog(null, "Enter first
integer:");
```

This results in the dialog box of Figure 2.4, with the user having typed "34" in the text entry area.

We then declare our integer variable `firstNumber`, and perform a conversion. The Java API provides a collection of methods related to the integer data type, and among these, there is a method to convert a `String` into an integer. The method `Integer.parseInt()` takes a single string parameter and converts the given string into an integer, returning the result. The following code accomplishes this for our example:

```
int firstNumber;
firstNumber = Integer.parseInt(userAnswer);
```

### 2.2.2 Dialog-based User Application

Let us put together what we have learned into a complete applicaiton.

#### Design

The following application will interact with the user by getting the user's name, and then retrieving two real numbers to add together. It should then compute the

sum and display the result in a user-personalized manner.

Our problem statement:

> Add two real numbers input by the user, and print the computed sum, using a personalized approach.

can be realized by the following steps:

1. Ask the user for their name, and retrieve the resulting string, $s$.
2. Prompt and retrieve the first real number, denoted $x$.
3. Prompt and retrieve the second real number, denoted $y$.
4. Compute $z = x + y$
5. Display the string `"Hi "` followed by the value of $s$, followed by `", your sum is "` followed by the value of $z$.

**The Program**

---

```
1  /**
2   * The AddDoubles class adds two real numbers (doubles)
3   * together, demonstrating user interaction through
4   * dialog boxes.
5   */
6
7  import javax.swing.JOptionPane;  // Tell Java where to find
8                                   // JOptionPane
9
10 public class AddDoubles {
11
12   // The single method of the class, main.
13
14   public static void main(String[] args) {
15
16     double sum;          // sum of firstNumber and secondNumber
17     double firstNumber;  //
18     double secondNumber; //
19
20     String userName;     // user entered name
21     String userAnswer;   // string with number to be converted
22
23     // Get user name
24
```

```
25      userName = JOptionPane.showInputDialog(null,
26                                              "Enter your name:");
27
28      // Retrieve the two numbers from the user
29
30      userAnswer = JOptionPane.showInputDialog(null,
31                                      "Enter first real number:");
32      firstNumber = Double.parseDouble(userAnswer);
33
34      userAnswer = JOptionPane.showInputDialog(null,
35                                      "Enter second real number:");
36      secondNumber = Double.parseDouble(userAnswer);
37
38      // Compute the sum
39
40      sum = firstNumber + secondNumber;
41
42      // Display result in a friendly way
43
44      JOptionPane.showMessageDialog(null,
45                      "Hi " + userName + ", your sum is " + sum);
46    }
47  }
```

# Chapter 3

# Arithmetic

Computers are well known for their fast and accurate arithmetic computations. The Java language provides support for arithmetic using notation familiar to all of us. For example:

```
1   int x = 5;
2   int y = 2;
3   int z = 4;
4   z = z/y;
5   x = x*y + z;
6   y = x*(y + z);
```

In lines 1 - 3 of this example, we have introduced the idea of initialization in Java. When a variable is declared, the programmer may immediately assign a value to it on the same line as the declaration. On line 1, we have not only declared `x` to be an integer, but we have assigned it the value of 5. Good programming dictates that it is a good idea to initialize every variable before you use it.

Using the Interaction window in DrJava, you can type lines 1 - 3 of the example code and then write:

```
1   System.out.println(z);
2   System.out.println(x);
3   System.out.println(y);
```

to get an immediate result of what the code does in the initialization lines. You can then type lines 4 - 6 followed by System.out.println statements for each of the variables again to see what happens as a result of those operations. It is also a good idea to try out a variety of assignments to help understand what results can

be obtained using addition, subtraction, multiplication, division, and parentheses.

In the example, variables named `x, y`, and `z` are declared to be integers and are assigned initial values. Note that the equal sign, `=` is used in Java to mean assign to. This is different from the meaning of the equal sign in mathematics. In an assignment, `x = 5;`, the meaning is not that x is equal to 5, but rather that x should be assigned the value 5.

On the fourth line, the value of `z` divided by `y` is assigned to the variable `z`. On the fifth line, the variable `x` is assigned the value obtained by first multiplying the values of x and y and then adding the value of z to the result. Notice that we use the `*` symbol to indicate multiplication, since writing xy would be indistinguishable from a variable named xy. Finally, on line 6, the variable `y` is assigned the value obtained by first adding the values of y and z and then multiplying the result by the value of x. This is exactly the way we would indicate this operation in mathematics.

Just as in standard mathematical practice, any computations enclosed within parentheses are carried out before other computations. After parentheses, multiplication and division are done in the order in which they appear, and finally, addition and subtraction are completed in the order in which they appear. These rules about what operations should be done before which others are called the rules of precedence in mathematics. Figure 3.1 lists operations in the order in which they are performed in the left column. The right column contains rules about where to start the evaluations and the left column tells whether to proceed from right to left or left to right or inside out to do the evaluations. For example,when evaluating expressions enclosed in parentheses, when some expressions are nested inside others, the rule requires that whatever is in the innermost parentheses should be done first.

Consider a more complex expression: 3*(-5 + (2 - 6)*2) -10%3 + 8/3. Using the table, we see that we need to start by evaluating whatever expression(s) lie inside parentheses. Moreover, we should start with the innermost parentheses. In this case, we need to evaluate the (2 - 6) first, resulting in -4. We now need to evaluate the newly formed expression 3*(-5 + -4*2)-10%3+8/3. Next we evaluate the expression -5 + -4*2, which requires us to first multiply -4 by 2 and then add the -5 to the result, yielding -5 - 8 which is -13. Now all the parentheses have been processed and we continue by carrying out multiplication, division, and taking the remainder as they appear, moving from left to right on the updated expression 3* -13 - 10%3 + 8/3. So we first multiply 3 by -13 getting -39. Next we take the remainder when 10 is divided by 3, which yields 5, and then dividing 8 by 3 we get 2. Note that all of the values in this example are integer and so the division omits anything that is not an integer. Now we have the expression -39 - 5 + 2 and

| Operator | Associativity |
|---|---|
| ( ) | inside out |
| unary - | right to left |
| *, /, % | left to right |
| +, - | left to right |

Figure 3.1: Operator Precedence and Associativity

following the table, we complete the subtraction and addition moving from left to right yielding -44 + 2 which equals -42.

## 3.1 Types

When writing mathematical expressions or formulas, we often use a variety of number types, such as natural numbers (non-zero only integers such as 3, 10, or 357), integers (whole numbers both postive and negative such as -25, 78, or 0), rational numbers (written as fractions or decimals such as 1/4, .25, or 234.5678), and real numbers (written in decimal notation such as 123.67, 1.3333, or 3.14). We use the same operation symbols when we mix these different numbers in the same expressions or formulas. For example, we might write the expression 3*(4.5 - $\pi$). Here we are multiplying an integer by the difference between a rational number and a real number. Human beings have no problem sorting this out and figuring out that since all of the numbers involved are in the real number set, the answer will be a real number, as opposed to an integer or rational.

Division poses an interesting phenomenon. Suppose we have two integers, 12 and 10. If we divide 12 by 10, the answer is not an integer. We can get an approximation to the answer, namely 1, but the exact answer is the rational number 1.2.

Although people automatically reason about such situations and make their own choices about whether they are satisfied with approximations, exact answers, or at least a more precise answer, computers need people to indicate their choices. For this reason we make explicit distinctions among these various mathematical types and use the type names to indicate what we want.

We will be able to write most of our Java arithmetic using two types: a type called `int`, standing for integer and a type called `double` to use when we want real numbers. The term ”double” has to do with precision, the number of digits on

the right hand side of the decimal point. Here are some instructions to try out using ints and doubles:

```
1  int x = 8;
2  int y = 6;
3  int quotient1 = x/y;
4  System.out.println(quotient1);
5  double z = 8.0
6  double w = 6.0;
7  double quotient2 = z/w;
8  System.out.println(quotient2);
```

On the third line, the integer variable `quotient1` is assigned the result of dividing the value of `x` by the value of `y`. Note that the result turns out to be 1. This is because both `x` and `y` are integers and so when you divide their values, you lose all but the integer part of the answer. On line 7, the `quotient2` is assigned the result of dividing the value of `x` by the value of `y` again, and this time all the variables have been declared to be of type `(double)` and so the value includes the decimal part, yielding 1.3333333333333333.

The next question we want to explore is ”What happens when an expression has a mixture of types. What type will the answer be?

Here are the rules Java uses to address the issue of what type the evaluation of an expression `a` <op> `b` will be. By <op> we mean some operation such as +, -, *, /, or %. Examples illustrating the rules will follow.

1. If a and b are of the same type, the result is of that same type.

2. If a and b are different types the operand is promoted to the greater operand and then the evaluation is done.

3. Java does not permit assignment of a greater type to become a lesser type.

4. Promotion of a lesser type to a greater type will happen automatically when an assignment is called for.

When we talk about lesser and greater, we refer to the following progression from lowest to greatest: `char`, `int`, `float`, `double`.

Here are some examples illustrating the rules:

1. `int x = 3; int y = 4;`
   Here `x + y` is the integer 7 Since both `x` and `y` are the same type, `int`, the sum is of type `int`.

2. `int x = 3;`
   `double y = 4.5;`
   If we want to add these, x + y, first x is promoted to double by rule 2, and then the addition takes place, resulting in the double 7.5. This example illustrates rule 2.

3. `int x = 0;`
   `double y = 5.2;`
   `x = y;` This is an error by rule 3.

4. `int x = 3;`
   `double y;`
   `y = x;` This is alright. y gets the value 3.0. This illustrates rule 4.

Sometimes we may have variables that we would like to treat as integers in some parts of a program, but as real numbers elsewhere. For example, we might get the ages of three people as whole numbers of years but then want to compute the average as a real number. Java allows us to convert between two different data types by an operation called "casting." To cast an integer as a double, we put the word `double` enclosed in parentheses in front of the integer. Here is an example:

```
1  int x = 8;
2  double y = (double)x;
3  int z = x;
```

We have declared an integer `x` and a double `y` and then assigned the value of `x` to `y`. In order to make the assignment, we cast the `x` to be treated as a double. Note that the variable `x` does not change to a double, so on line 3, the assignment puts the integer value of `x` into the integer variable `z`.

The type `int` can be used for integers between -2,147,483,648 and 2,147,483,647. Although these numbers are large enough for many computations, sometimes we

| Type | Value Range |
|---|---|
| boolean | true, false |
| float | $\pm 3.4 \times 10^{-38}$ to $\pm 3.4 \times 10^{-38}$ with 7 digits of accuracy |
| byte | -128 to 127 |
| int | -2,147,483,648 to 2,147,483,647 |
| char | ascii characters |
| long | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,808 |
| double | $\pm 1.7 \times 10^{-308}$ to $\pm 1.7 \times 10^{-308}$ with 15 digits of accuracy |
| short | -32,768 to 32,767 |

Figure 3.2: Primitive Types

may need even larger integers. For example, suppose we want to find out how many times a person's heart will beat during an average lifetime. Currently, life expectancy is 80 years and the average heartrate is 70 beats per minute. To get the number of beats in a lifetime, we need to multiply 80 by 70 by the number of minutes in a year. This multiplication results in a very large integer. To accommodate such large numbers, we can use a type called *long*.

```
1  long lifetimebeats = 0;
2  long minutesperyear = 0;
3
4  minutesperyear = 60*24*365;
5  lifetimebeats = 70 * minutesperyear * 80;
6  System.out.println("The average number of times a heart beats in a l
```

There are additional types in Java that we may use later. In Figure 3.2, we present a table of several types in Java. We will introduce examples of types throughout the text as we need them.

## 3.2 Special Math Methods

Some mathematical computations require more than just a simple sum or product. For example, suppose we want to solve a quadratic equation $x^2 + 5x - 3$. One way to do this is to use the quadratic formula solution which requires us to take the square root of $25 - 4 * 2 * (-3)$. The formula solves the general equation $ax^2 + bx + c$ is $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$.

Java provides commonly needed mathematical functions implemented as methods collected together into a class called Math. We can use those methods in the programs we write by importing the Math class at the beginning of our program and then calling the methods such as `sqrt`, to suggest square root. The * in Line 5 of the program indicates that we want all of the available methods in the class called *java.math*. To indicate that the method comes from the Math class, we write `Math.sqrt()` on the line of the program in which we use the square root method. On Line 13 we assign the value that `Math.sqrt()` computes to the variable solution. This can happen because the method called `Math.sqrt()` produces a value consistent with the type of the variable `solution`. Example:

```
1  /**
2   * This program computes part of the quadratic formula
3   */
4  //get the Math functions to use.
5  import java.math.*;
6
7  public class MathUsage
8  {
9      public static void main(String[ ] args)
10     {
11         double solution = 0;
12         // factor: 0 = xˆ2 + 5x - 3
13         solution = Math.sqrt(25 - 4*2*(-3));
14         System.out.println(solution);
15     }
16 }
```

Some other mathematical operations that Java provides methods for include `abs()` for absolute value, the typical trigonometric functions, `sin(), cos(), tan()`, as well as functions to compute the maximum and minimum, `max()` and `min()` respectively. There is also a function that produces a random value, `random()`.

## 3.3 Building Expressions

We have seen how various expressions are written and evaluated according to rules of precedence and associativity by looking at examples. We now want to see the rules for how to build a larger expression from smaller components. At the simplest

level, expressions can be constants, variables, or method invocations. An example of a constant is `3`. We have seen and used many variables with names such as `x`. In our quadratic formula we used the method invocation `Math.sqrt()`.

In formal notation, letting `<E>` represent an expression, we say this as follows:

```
<E> := <constant> | <variable> | <method invocation>
```

The `|` mark means "or."

Using the same notational forms, we can show how other expressions can be built:

```
<E> := (<E>) | <E> + <E>  |  <E> - <E>
          |  <E> * <E>  |  <E> / <E>
```

In other words, we can build complicated expressions from simple ones using parentheses and arithmetic operator symbols.

**Exercise 3.1** *Write a progam to compute a student's gpa. You should allow the user to input a grade between 0.0 and 4.0 for each of four courses and then compute the gpa and print it out in an appropriate message. The input grades should be between 0 and 4.0.*

**Exercise 3.2** *Write a program to find the weekly earnings for an employee. Ask the employee what the hourly wage is and how many hours the person has worked. Print out the earnings in appropriate form with a dollar sign.*

**Exercise 3.3** *Write a program that inputs a number, either positive or negative, and prints out the absolute value.*

**Exercise 3.4** *Write a program that inputs a number of degrees in Fahrenheit and prints out the number of degrees in Celsius. To get this value you need to multiply 5/9 by (F - 32) where F means the number of degrees in Fahrenheit.*

**Exercise 3.5** *Write a program that inputs ages of three people and then prints out the average age. Declare the ages to be integers, but compute the average to include the fractional part.*

# Chapter 4

# Conditionals

A lot of what computers do for us is what we call “event driven,” meaning that the computer responds to some external event such as an input by a user. Of course, what the computer should do upon receiving an input usually depends on what that input is. For example, if you are getting input via a dialog box and you ask the question “Do you want to continue with this program?”, then a negative reply means that the user wants the program to stop, but if the user says“yes,” you want the program to continue doing whatever it has been written to do. Here, a decision has to be made based on an input value.

Sometimes a decision must be made based on a computation. For example, in the previous chapter, Expressions, we computed the number of hearbeats for a person who lives an average lifespan. Suppose we want to provide for those who are interested, not only the number of heartbeats, but also the number of minutes in the average lifespan. We can ask the user to choose whether or not he/she wants to know the number of minutes as well as the number of heartbeats. Depending on the response, the program will provide both statistics or only the number of heartbeats. Look at the example that computes number of heartbeats and think about how you could print the number of minutes for those users who request it, but omit that information for those who do not. Using only sequential control for what happens, we have no way to skip some statements, so we either give every user both values or we don’t. Before modifying this program to satisfy the requirement that we provide number of minutes only for those who want to see it, we need to find out how Java allows code to be skipped.

Java supports decision making by what are called “conditional” statements and by “switch” statements. The conditionals are usually used when there are only two choices and the switch when there are several choices. However, either can be used whenever different actions are associated with some result.

## 4.1 if Statements

An `if` statement takes the form:

```
if ( <boolean expression> )
{
   <stmt>
}
<stmt>
```

The first statement sequence, `<stmt>`, is executed only when the boolean expression evaluates to `true`. It is skipped otherwise. The second statement sequence is carried out in either case.

Here is the modified example for finding real solutions to quadratic equations.

A typical Boolean expression is a relational expression that has two parts separated by a comparison symbol and evaluates to either "true" or "false." Here are some examples:

1. x < 3
   If `x` has a value less than 3, the expression evaluates to "true." If `x` has value 3 or greater, the value of this expression is "false."

2. 4 > 7
   Note that this expression will always evaluate to "false." This type of statement doesn't usually appear in a program since it is always false.

3. 500 <= x + y
   The value depends on `x` and `y`.

4. x == y
   This syntax evaluates to true when `x` and `y` have equal values.

5. w != y
   This checks to see if `w` does not equal `y`.

6. -456 >= x*y - w
   Here some computation must be done in order to determine the truth value of the expression.

| | |
|---|---|
| < | less than |
| > | greater than |
| <= | less than or equal to |
| >= | greater than or equal to |
| != | not equal |
| == | equal |

Figure 4.1: Symbols and Meanings

Figure 4.1 is a list of the possible relations and their meanings:

When used in an expression each of these symbols result in an evaluation to "true" or "false." We call the type whose values are "true" or "false" `boolean`, one of the primitive types in Java. Each boolean expression has a value of this type. Notice the "equals comparison" has two equals signs `==`. A single equals denotes an assignment statement, you must use the double equals for comparing two items in a boolean expression.

We are now ready to modify the program that computes heartbeats so that it also prints number of minutes for those who want to know both:

```
1  /**
2   * This program computes the average number of times a heart beats
3   * in a lifetime.  For those who want to know, it also displays the
4   * number of minutes in an average lifespan.
5   */
6
7  import java.util.Scanner;
8
9  class HeartBeats
10 {
11     public static void main(String[] args)
12     {
13         long lifetimebeats = 0;
14         long minutesperyear = 0;
15         long minutesperlife = 0;
16         int response = 0;
17
18         minutesperyear = 60*24*365;
19         minutesperlife = 80 * minutesperyear;
```

```
20          lifetimebeats = 70 * minutesperlife;
21
22          System.out.print(
23              "The average number of times a heart beats");
24          System.out.println(
25              " in a lifetime is " + lifetimebeats + ".");
26          System.out.print(
27              "Would you like to know the number of ");
28          System.out.println(
29              "minutes in an average lifespan?");
30          System.out.println("Enter 1 for yes and 0 for no");
31
32          Scanner consoleIn = new Scanner(System.in);
33          response = consoleIn.nextInt();
34
35          if(response == 1)
36          {
37              // This output occurs only when the response is 1.
38              System.out.print(
39                 "The number of minutes in an average lifespan");
40              System.out.println(" is " + minutesperlife + ".");
41          }
42      }
43  }
44
```

This program computes both the number of minutes in an average lifespan and the number of heartbeats. However, after it prints the number of heartbeats, it asks the user whether he/she also wants to know the number of minutes. If the repsonse is in the affirmative, the program provides the number of minutes, but if the response is negative, the output line is skipped and the program ends.

Here is another example of the `if` statement:

```
1  if (sales > 75000)
2  {
3      bonus = 1000;
4      System.out.println("Your bonus is $1000.");
5  }
6  System.out.println(
7     "The target for next month is $75000.");
```

In this code segment, the conditional statement checks to see if the sales value is greater than 75000. If it is, a message is printed out awarding a bonus. Whether or not the sales value exceeds 75000, a message is printed telling what the target for next month will be. Notice that the steps which must be executed when the boolean expression evaluates to `true` are enclosed within a pair of braces. When there is only one statement to be done, the braces are not required, but for any number of statements greater than one, the braces are needed.

## 4.2 if else Statements

Often, there are two distinct actions desired upon evaluating a boolean expression, one action to take place if the expression is true and another if it is false. For example, if we want to find the larger of two numbers, we can compare one to the other and then choose to output the larger one based on the results of the comparison:

```
1  int x = 5;
2  int y = 7;
3  if (x > y)
4  {
5      System.out.println(x + " is larger than " + y);
6  }
7  else
8  {
9      System.out.println(y + " is larger than or equal to " + x);
10 }
```

In this program segment, the value of `x` is compared to `y`. Since `x` fails to be greater than `y`, the control flow will cause the program to skip the first output line and go to the line following the **else** part.

The form of an `if else` statment is:

```
if ( <boolean expression> )
{
   <true stmt block>
}
else
{
```

```
    <false stmt block>
}
```

We are now ready to modify the program we did for getting solutions to quadratic equations by checking to see whether the solutions are real:

```
1  /** This programs allows users to enter coefficients for the
2   * quadratic equation a*x^2 + b*x + c.  If the solutions are
3   * real, the program computes them and prints them, but if
4   * the solutions are not real, the prpogram prints an
5   * appropriate message for the user.
6   */
7
8  import java.math.*;
9  import java.util.Scanner;
10
11 class QuadSolutions
12 {
13     public static void main(String[ ] args)
14     {
15         double a, b, c = 0.0;
16         double solution1, solution2 = 0.0;
17
18         Scanner consoleIn = new Scanner(System.in);
19         System.out.println("Enter your value for a:  ");
20         a = consoleIn.nextDouble();
21
22         System.out.println("Enter your value for b:  ");
23         b = consoleIn.nextDouble();
24
25         System.out.println("Enter your value for c:  ");
26         c = consoleIn.nextDouble();
27
28         if (b*b - 4*a*c < 0.0)
29             System.out.println(
30             "Your solutions are not real numbers.");
31
32         else
33
34         {
```

```
35              solution1 = (-b + Math.sqrt(b*b - 4*a*c))/2*a;
36              solution2 = (-b - Math.sqrt(b*b - 4*a*c))/2*a;
37
38              System.out.println("The solutions are  "
39                + solution1 + "  and  " + solution2);
40          }
41      }
42  }
43
```

In this modified version of the quadratic program we set up variables to hold the values of the coefficients of the quadratic $ax^2 + bx + c$ and variables to hold the solutions if they exist. Before doing the computation the program checks the value of the square root to see if it might be negative. If it is negative, the program produces a message indicating that the solutions are not real. But if the value of the square root is not negative, the program computes both roots and prints them out.

## 4.3 Compound Boolean Expressions

Sometimes there may be more than one condition controlling what action is to be taken. For example, in a particular company, it may be that anyone who makes more than 10 sales or who sells a total amount of at least $85,000 will receive a bonus of $2,000. In this case we need to check two expressions indicating that if either one or the other evaluates to true, the salesperson gets the bonus. To express an *or* we use the symbol ||.

```
1   double totalSales = 0.0;
2   int numberSales = 0;
3
4   // insert code here to enter values for
5   // totalSales and numberSales
6
7   if (totalSales >= 85000 || numberSales > 10)
8   {
9       System.out.println("You get a bonus of $2,000.");
10  }
11
12  // the rest of your program goes here
```

We have omitted parts of the program to retrieve input for `totalSales` and `numberSales`.

| x | y | x \|\| y | x && y |
|---|---|---|---|
| T | T | T | T |
| T | F | T | F |
| F | T | T | F |
| F | F | F | F |

Figure 4.2: Evaluation of Compound Boolean Expressions

If you need to confirm that more than one condition must evaluate to true in order for certain code to be executed, we use the symbol &&. For example, suppose that students who are under 19 and who have at least a 3.0 average are eligible for the junior debate team. We can indicate this as follows:

```
1  if ( age < 19  &&  gpa >= 3.0 )
2  {
3      System.out.println(
4       "You are eligible for the debate team.");
5  }
```

The rules for how to evaluate compound statements are shown in the table, Figure 4.2.

## 4.4 switch Statements

It is possible by using multiple `if else` combinations to handle situations that involve several actions depending on the evaluation of one or more boolean expressions. For example, depending on what year of college a particular student is in, a different status will be assigned. Assuming a student inputs the value of year as 1 or 2 or 3 or 4, the following code illustrates how status would be noted:

```
1  switch(year)
2  {
3      case 1:
4          System.out.println("Freshman.");
5          break;
6      case 2:
7          System.out.println("Sophomore.");
```

```
8           break;
9       case 3:
10          System.out.println("Junior.");
11          break;
12      case 4:
13          System.out.println("Senior.");
14          break;
15      default:
16          System.out.println("Not a valid year.");
17  }
```

The word `switch` is a keyword in Java. The word *year* in the parentheses is the variable name on whose value the control flow depends. `case` is a keyword followed by a particular value that *year* might receive as input. For each case, some statement or sequence of statements following that case indicate what action(s) must be taken for that case. The keyword `break` is used to terminate the switch statement. This is important here, so that once a particular case has been determined, the correct action is taken and then the control flow passes to the next statement following the switch statement. Without the `break`, control would pass through all the cases, evaluating them all. The last case listed is one called `default`. This allows the programmer to handle the situation when the user has given as input some value that none of the other case statements has. In this example, in the event that a user gave any value other than the four values allowed, a message will make that clear to the user. It is important to note that the value on which the `switch` statement depends must be discrete, such as `int, char, long, byte, boolean, and short` not a value which might need to be an approximation, such as a `double`. For example, it's alright to make the `switch` statement depend on an integer, but not on a real number (double in Java).

## 4.5 Nested Conditionals

There may be more than one way to make sure your program follows the control sequencing needed. For example, instead of using a switch statement for printing out a message telling whether a student is a Freshman, Sophomore, Junior, or Senior, it is possible to use a series of if-else combinations. One of the exercises in this chapter asks you to do so.

For more complicated situations, one might consider placing one `if` statement inside the body of another `if` statement or `if else` statement to express what

needs to happen in a convenient way. This is called "nesting." Let's consider the following problem:

A bank wants to screen potential borrowers online by asking them some questions to see if they qualify for a loan before making an appointment to spend time discussing loans with the customer. The bank requires that a person be employed in order to be eligible for a loan, but the bank also wants the potential borrower to earn at least $25,000 per year. If a person fails to meet the employment requirement, the bank wants the person to receive a message saying that a person must be employed in order to be considered for a loan. If the person is employed but fails to meet the $25,000 requirement, the bank wants the person to receive a message telling the person that their income is not high enough.

Let's follow the steps we saw in the first chapter for preparing a program for the bank. For the design stage, we have a problem statement in the previous paragraph that describes what the bank wants the program to accomplish. Next we need to make a sequence of statments that will satisfy those requirements.

1. Prepare a message to display to potential borrowers telling them what information they will need to provide and in what form they should provide it. Let's plan to tell the customers that they will need to tell whether or not they are employed and whether or not their income exceeds $25,000.

2. Ask whether the customer is employed and provide a variable in which to store the responses.

3. Ask whether the customer has an income above $25,000 and provide a variable in which to store the respone.

4. Check to see if the customer is employed.

    - If the customer is employed, then check to see if the income is adequate.
    - If the customer is employed, but the income is not adequate, display a message saying so.

5. If the customer is not employed, display a message saying so.

Let's look at what code would be entered when you use your editor to write the code. We have chosen to call our program class "LoanQualification." In the main method we have declared two integers `employed` and `income` and initialized them to 0. On line 11, we alert the user to answer questions using a `1` for an affirmative answer and `0` for a negative answer. The first question concerning

employment is output on line 13. On line 14 the variable `employed` is assigned the input.

Line 16 asks about income and on line 17 `income` is assigned whatever the user enters.

On line 19 we begin our first `if` statement. The expression to be evaluated checks the value of `income` to see if it is equal to `1`. If so, another `if` statement is introduced to see if the `income` value is `1`. We call this a "nested" `if` because the second `if` is processed only when the first `if` expression evaluates to `true.`

When the first boolean expression (`(employed == 1)`) evaluates to `true` but the second one (`income > 25000)` ) is `false`, the `else` section is executed, producing the message about income.

Note that if the user inputs a `0` indicating that the user is not employed, then the income check does not occur, but rather the control jumps to line 32, producing the message about the need for employment.

The use of the nesting is a convenient way to produced the desired results. It is important to note that most conditional situations may be achieved in a variety of ways, some more cumbersome than others. It is a good idea to give some thought to what you need to accomplish to determine which of the possible conditional statements would suit your situation best.

```
1  /**
2   * This program determines whether a potential lender
3   * qualifies for a bank loan or not.
4   */
5  import java.util.Scanner;
6
7  class LoanQualification
8  {
9      public static void main(String[ ] args)
10     {
11         int employed, income =0;
12
13         // retrieve user data from keyboard input
14         Scanner consoleIn = new Scanner(System.in);
15
16         System.out.println(
17            "Answer the following questions with 1 " +
18            "for yes and 0 for no.");
19
20         System.out.println("Are you employed?  ");
```

```
21          employed = consoleIn.nextInt();
22
23          System.out.println(
24             "Is your income above $25,000 per year?  ");
25          income = consoleIn.nextInt();
26
27          // We check to see if the user is employed.
28          if (employed == 1 )
29          {
30              // Only if the user is employed do we check
31              // whether the income is adequate.
32              if (income == 1 )
33              {
34                  // If both expressions are true,
35                  // the user qualifies for the loan.
36                  System.out.println(
37                     "You qualify for a bank loan.");
38              }
39              else
40              {
41                  // the user's income is inadequate.
42                  System.out.println(
43                     "Your income is not high enough " +
44                     "to secure a loan.");
45              }
46          }
47          // only print this message if user is unemployed.
48          else
49          {
50              System.out.println(
51                 "You must be employed to qualify " +
52                 "for a loan.");
53          }
54      }
55  }
```

## 4.6 Exercises

**Exercise 4.1** *Write a code segment that inputs an amount representing a salesperson's total sales for a month. If that input is greater than 75000, the program should print a bonus of 1000. In any case, the program should print a message encouraging the salesperson to work toward next month's sales.*

**Exercise 4.2** *Write a method that inputs two numbers and outputs the smaller of the two.*

**Exercise 4.3** *Write a program that asks the user to input the number of hours worked during the past week and the hourly rate of pay. If the number of hours is 40 or less, the amount earned is that number of hours multiplied by the rate. If the number of hours is greater than 40, then the amount earned is 40 times the rate plus time and a half for the number of hours over 40. The program you write should print out an appropriate message that includes the amount earned.*

**Exercise 4.4** *Write a program that asks for a child's name and age and the child's readiness score. If either the age is greater than 6 or the readiness score is greater than or equal to 85, the program should print a message stating that the child is ready for first grade. Otherwise, there should be a message stating that the child should try again at a later time.*

**Exercise 4.5** *Write a program that inputs two real numbers and then allows the user to ask for any of the following: the sum, the difference, the product or the quotient of the numbers. Hint: You may want to have the user type the letter "S" for sum, "D" for difference, etc. To do this you will need to use a type called* **char** *that allows variables to have ascii values.*

**Exercise 4.6** *Wrtie a program that allows a user to input 3 grades between 0 and 100. The program finds the average and outputs both the numerical average and an appropriate letter grade. Assume that grades are assigned on a 10 point scale where below 60 is "F," 60 to 69 "D," etc.*

**Exercise 4.7** *Write a program that inputs a student's year as an integer and then prints out Freshman, Sophomore, Junior, or Senior depending on whether a 1 or 2 or 3 or 4 is entered. Use a succession of* `if` *or* `if else` *statements, rather than a switch statement for the purpose of showing that it can be done.*

# Chapter 5

# Repetition

We have seen how it is possible to transfer control within a program by evaluating a boolean expression and then either executing or skipping certain code segments. Sometimes it is particularly useful to control the flow of action in a program by repeating some collection of statements over and over again until some task has been accomplished. For example, suppose you want to find the largest value among four numbers. You could declare four variables and assign a value to each and then start using conditional statements to compare those numbers in an attempt to find the largest. However, such a program would not only be cumbersome to write, even worse, it would defeat the purpose of automating the task at all, since it would probably be quicker to just process the numbers without using a computer at all.

Our code would like this this:

```
1   /**
2    * This program finds the largest in a set of 4 integers
3    * using a different variable for each integer.
4    */
5
6   import java.util.Scanner;
7
8   class FindLargest
9   {
10    public static void main(String[ ] args)
11    {
12      int a, b, c, d = 0;
13      int largest = 0;
14      Scanner consoleIn = new Scanner(System.in);
```

```
15
16      System.out.println("What is your first number?");
17      a = consoleIn.nextInt();
18
19      System.out.println("What is your next number?");
20      b = consoleIn.nextInt();
21
22      System.out.println("What is your next number?");
23      c = consoleIn.nextInt();
24
25      System.out.println("What is your next number?");
26      d = consoleIn.nextInt();
27
28      largest = a;
29
30      if(b > largest)
31        largest = b;
32      if(c > largest)
33        largest = c;
34      if(d > largest)
35        largest = d;
36
37
38    System.out.println("The largest is   " + largest);
39    }
40  }
```

Even with four numbers to process, the program gets long and repetitious. Think of what would need to be done if we needed to find the largest among 100 numbers or more. Next we will see a way to find the largest among 100 numbers by using far fewer than 100 variables by using a new construct called a "loop." There are several kinds of loop constructs. We will examine two of them. The first we will consider is the `while` loop.

The syntax for the `while` loop is given by:

```
while  ( <boolean expression> )
{
   <stmt>
}
```

Semantically we note two parts of the construct: First, there is the question of

how many times the body of the loop should be executed. That is determined by the boolean expression. The second part is the code to be iterated (repeated). This is called the “body” of the loop.

In the `while` loop, the boolean expression determines the flow of control. As long as that expression evaluates to `true`, the statements in the loop are executed over and over. The repetition stops only when the boolean expression evaluates to `false`. Here is an example of a code fragment that finds the largest among 10 integers (it would be trivial to modify the program to find the largest among 100 integers, but this takes too long if you want to actually test the program) :

```
1   /**
2    * This program finds the largest of 10 integers.
3    */
4
5   import java.util.Scanner;
6
7   class FindLargest10
8   {
9       public static void main(String[ ] args)
10      {
11          // holds one number at a time as it is read.
12          int number = 0;
13          // holds the largest number entered so far.
14          int largest = 0;
15          // keeps track of how many have been read.
16          int counter = 1;
17
18          Scanner consoleIn = new Scanner(System.in);
19          System.out.println("What is your first number?");
20          // note: first input is automatically the largest
21          // number seen so far
22          largest = consoleIn.nextInt();
23
24          while (counter < 10)
25          {
26              System.out.println("Enter next number: ");
27              number = consoleIn.nextInt();
28              if (number > largest)
29                  largest = number;
```

```
30              counter = counter + 1;
31          }
32          System.out.println("The largest is "
33             + largest + ".");
34      }
35  }
```

When control reaches the `while` construct, the boolean expression is evaluated. When this point of the program is encountered for the first time, the value of `counter` is 1 and so the boolean evaluates to `true`. This means that the statements between the braces will be executed once. During the execution of the loop body, a number is input and assigned as the value for `number`. That value is compared to the current value of `largest` and if the new number is bigger than the current value of `largest`, the value of `largest` is changed to be whatever the newly input value of `number` is. Note that before the loop begins, the first of the 10 numbers is read into the variable called *largest* and so when the loop has executed once, the larger of the first two numbers is already stored in the variable called `largest` and so at that point, the value of `largest` is the biggest of the values because it is the only value. The last statement in the loop increments `counter` by 1, making it 2. At the end of the loop code as indicated by a brace, the boolean expression is again evaluated. Of course, since the value of `counter` is now 2 and is still less than 10, the loop executes again. So a third number is input and compared against the value of `largest`.

Remember that `largest` holds the larger of the first two numbers, so if the value of `number` is still bigger, it will replace `largest`. If not, `largest` will remain as is, now being the maximum of the first three numbers. This process continues. Each time the loop is executed, `counter` increases, the boolean expression is checked, and as long as the value of `counter` remains less than 10, the loop steps are executed. However, once the counter reaches 10, this means all the numbers have been read. Now the boolean expression evaluates to `false` and the loop steps are skipped entirely, bringing control to the output statement. Since the value stored in the variable `largest` is the largest of all 10 numbers, it is printed out.

## 5.1 `for` Loops

When the programmer knows exactly how many times a code section must be repeated, an alternative to the `while` loop is a construct called a `for` loop. A second example finds the largest number among 10 numbers entered using a `for` loop, the same job we did previously with a `while` loop.

```
1  /**
2   * This program finds the largest of 10 integers.
3   */
4
5  import java.util.Scanner;
6
7  class FindLargestForLoop10
8  {
9      public static void main(String[ ] args)
10     {
11         // holds one number at a time as it is read.
12         int number = 0;
13         // holds the largest number entered so far.
14         int largest = 0;
15         // keeps track of how many have been read.
16         int counter;
17
18         Scanner consoleIn = new Scanner(System.in);
19         System.out.println("What is your first number?");
20         // note: first input is automatically the largest
21         // number seen so far
22         largest = consoleIn.nextInt();
23
24         for (counter = 2; counter <= 100; counter++)
25         {
26             System.out.println("Enter next number: ");
27             number = consoleIn.nextInt();
28             if (number > largest)
29                 largest = number;
30         }
31         System.out.println("The largest is "
32            + largest + ".");
33     }
34 }
```

In this example, the code section preceding the loop is the same as we used earlier. We simply set up variables to use for keeping track of the number currently under consideration, the variable used to store the largest number, and a variable to use for counting how many numbers have been read in. As before we read in the first number and immediately identify it as the largest, since it is the largest so far.

In the loop, the keyword `for` is followed by a pair of parentheses in which there are three parts. The first part, the loop initialization, tells where to start. In this case we have chosen to start counting at 2, since we already read in the first number. The second part, the boolean condition, tells what boolean expression should be checked each time the loop is entered. If this boolean condition is true, the loop will be entered again. It will continue to be entered until the condition is false. In this particular case we want the program to continue reading numbers until 10 have been read. The third part, the loop update, tells how to keep the counter variable up to date. This update statement is always executed at the end of every loop iteration. Note that `counter++` is a shorthand way of writing `counter = counter + 1` but means exactly the same thing.

The code segment to be repeated is enclosed between braces just as in the while loop. Note that we do not need to increment the counter in the repeated segment, since the loop update part of the `for-loop` inside the parentheses takes care of that.

An obvious question concerns when to use a `while` loop and when to use a `for` loop. It turns out that any loop can be written with either construct, but there is a definite convention that should be followed. `for` loops should be used any time that you know *the exact number of loop iterations* **before** entering the loop. Otherwise, you should use a `while` loop construct. In our largest-among 10 integers problem, a `for` loop is the correct choice because before the loop executes, we already know that we need exactly 10 iterations. However, if we were to prompt the use to keep entering numbers until they enter a 0, we would need a `while` loop since the exact number of interations is not known – it depends on what the user enters inside the loop body.

## 5.2 Nested Loops

Just as it was possible and sometimes convenient to put conditional statements inside other conditionals, there are times when we will need to put loops insider other loops. Before doing an application of this technique, it is useful to see what happens when we write a program that illustrates what happens when one loop is nested inside another.

```
1  /**
2   * An example of nested loops.  Can you
3   * guess what is printed?
4   */
5  class NestedLoop
```

```
6   {
7       public static void main(String[ ] args)
8       {
9           int i,j = 0;
10
11          for (i = 0; i < 4; i++)
12          {
13              System.out.println("i is " + i);
14              for (j = 0; j < 3; j++)
15              {
16                  System.out.println("j is " + j);
17              }
18          }
19      }
20  }
```

This short program serves to show how a loop inside another loop behaves. We often call the first loop (with the `i` loop control variable) the *outer loop* and the second loop (with the `j` loop counter) the *inner loop*. When the outer loop is first reached, the value of the loop counter `i` is set to 0, the boolean expression is evaluated, and since the value is true, the inner loop is reached. Its control variable, `j` is set to 0, the boolean expression is evaluated, and since the value is `true`, the body of the second loop is executed. At the end of one iteration of the inner loop, `j` is incremented and the boolean expression evaluted with the new value of `j`. Since that value is still `true` the inner loop body is executed again. This repetition of execution, incrementing, and expression evaluation continues until the boolean expression finally evaluates `false` when `j` reaches 3.

It is only when the inner loop finishes that control reverts to the outer loop, incrementing the `i` (now to the value of 1), evaluating the boolean expression to see if `i` is less than 4 and then upon an evaluation of `true`, repeating the loop body once again. Since the loop body contains the inner loop, when control reaches the inner loop, the value of `j` is set back to 0 and the whole porcess begins again.

The result is that for each of the four iterations of the outer loop, the inner loop goes through 3 iterations. Run the program to confirm that you understand how the control flows. Try to guess the output before you enter and run the program.

## 5.3 Applicaton: A Multiplication Table

In this application we use one new idea besides nested loops. To provide flexibility so you could print a table for any number of values, not just 10, we introduce

the idea of a constant. The integer representing the highest value whose pairs we want the product for has the word final in front of it: `final int MAX = 9;`. Final means that the value is set in the declaration and can never be changed in the program. In this case we have set the value at 9, remembering that the digits go from 0 to 9. It is common convention to use all uppercase letters for a constant so that they are easy to identify in the code.

Problem Statement: Write a program that will produce a multiplication table for all 10 digits, showing the product for each pair of digits. Keep the program flexible so if we wanted a multiplication table for just 5's or 8's or any other number, we could just change the line of the program where we declared `MAX`.

Design:

1. Choose variables to represent the digit pairs whose product is required.
2. Choose a variable to represent the maximum digit for which we want the table of product pairs.
3. Display a heading for the table.
4. For each digit, find the product of it with every other digit and display the products on a single line labeled by that digit.

```
1  /**
2   * This program produces a multiplicaton table
3   * for pairs of digits.
4   */
5
6  class MultTable
7  {
8      public static void main(String[ ] args)
9      {
10         int i;
11         int j;
12         final int MAX = 10;
13
14         // print the top header row
15         System.out.print("* |  ");
16         for (j = 1;  j <= MAX; j++)
17         {
18             System.out.print( "  " + j + "  ");
19         }
```

```
20          System.out.println();
21          for (j = 1;  j <= MAX; j++)
22          {
23              System.out.print( "------");
24          }
25          System.out.println();
26
27          // print the table
28          for (i = 1; i <= MAX; i++)
29          {
30              System.out.print(i + " |  ");
31              for (j = 1; j <= MAX;  j++)
32              {
33                  System.out.print( "  " + i * j + "  ");
34              }
35              System.out.println();
36          }
37      }
38  }
```

## 5.4 Exercises

**Exercise 5.1** *Write a program that allows a user to input as many numbers as the user wants to enter and then outputs the number of values that were entered.*

**Exercise 5.2** *Write a program that asks how many numbers a user wants to add and then allows those numbers to be entered. After all the numbers are entered, the program outputs the sum.*

**Exercise 5.3** *Write a program that finds the average of however many numbers a user may want.*

**Exercise 5.4** *Write a program that allows a user to input as many numbers as the user wants to enter and then outputs the maximum and the minimum numbers among the entered numbers.*

**Exercise 5.5** *Write the same programs again using the other kind of loop from the one you used the first time.*

# Chapter 6

# Strings

We have been using examples of strings throughout our programs thus far. This chapter takes a closer look at the syntax and semantics associated with creating and manipulating the data type of `String` in the Java language.

We first note that the type `String` is not one of the primitive types listed in our table of Figure 3.1. We do see the type '`char`' in that table as having a value of ASCII characters. Intuitively, a `String` is a sequence of characters grouped together. For instance, the sequence of characters 'J', 'a', 'v', and 'a' can form the `String` `"Java"`. This more complex type has its own class definition, and is referred to as a *class type*. We shall learn to create our own class types in Chapter 9, but the focus of this chapter is on how to *use* the `String` class type to greater effect in our own programs.

We begin by reviewing some of the concepts from earlier in the text.

## 6.1 String Syntax and Semantics

### 6.1.1 String Constants

We have already seen that we can introduce a `String` constant into our program by simply putting the desired sequence of characters between double quotes. For instance, our first program used the constant `String`, `"Welcome to Java Programming!"` as a parameter to the `System.out.println()` method.

A string constant, just like a variable defined as (class) type `String`, is an expression whose type is `String` and whose value refers to the sequence of characters that make up the constant. As such, a string constant can be used in any case where a string expression is appropriate. We have seen this in the use of string constants as parameters to methods, such as `println` and on the right hand side

of assignment statements.

### 6.1.2 String Declarations and Initialization

In a manner similar to the declarations statement for the primitive data types of `int` or `double`, we can declare a variable to refer to a `String`. The syntax

```
String <identifier>;
```

is exactly the same as that introduced in Chapter 1. The declaration creates the variable <*identifier*> and associates storage in the memory of the machine with that <*identifier*>. It also associates the data type of `String` with <*identifier*>, constraining the valid values that may be stored in the variable.

We can also initialize a `String` variable when it is declared. The syntax is

```
String <identifier> = <string-expression>;
```

where the <*string-expression*> is most often a <*string-constant*>, as defined above. For example, we can declare and initialize `String` variable `first` and `second` as follows:

```
String first = "Now is the time ";
String second = "for all good persons";
```

and now I can use the variables `first` and `second` anywhere that a `String` is appropriate. We have already seen the use of `String` variables in the context of user interaction.

### 6.1.3 String Concatenation

We use the '`+`' operator with `String`s to form longer strings that are the concatenation of the `String` operands. Continuing our example from above, the string expression `first + second` results in a new `String` whose value is `"Now is the time for all good persons"`. This string expression could be assigned to a new `String` variable:

```
String third = first + second;
```

or it could be used as the `String` argument to an output statement:

```
System.out.println(first + second);
```

or anywhere else a string expression is appropriate.

The string concatenation operator in Java is also frequently used to help us convert numbers and other data types into `String`s. If *either of the operands* of a '`+`' operator is a string, then the other operand is automatically converted to a string as well, and then both strings are concatenated. We have been using this property from some of our earliest programs in Chapter 1. For example, suppose we have the following sequence of code:

```
int age = 25;
String prefix = "My age is ";
String myage = prefix + age;
```

In the third statement, since `prefix` on the right hand side of the assignment statement is a `String`, the integer `age` is converted from the value 25 to the two character string sequence `"25"`. Then the two strings `"My age is "` and `"25"` are concatenated together to form the string `"My age is 25"`.

String concatenation also works with non-numeric types. Suppose I want to combine a primitive type `char` with a `String`. I can use the same technique:

```
char letterS = 's';
String animal = "frog";
String plural = animal + letterS;
```

to yield the value `"frogs"` for the variable `plural`.

## 6.2 String Manipulation

When we increase the complexity of a data type, we often wish to manipulate and access some of the simpler types that make up the more complex type. This is certainly true for the class type `String`. For instance, one attribute of any string is its length. The length of the string `"Welcome to Java Programming!"` is 28. Note that we include the characters that are spaces and punctuation, ... , they are part of the string. The quote marks, however, are not. They are part of the syntax that we use to denote a string constant.

When a class type is defined, the designer of that type may define methods that operate on instances of this type so that we can query and manipulate the constituent attributes of the complex type. For instance, the class type of `String` has a method called `length()` that returns the length of a particular `String` instance. We invoke a method for a particular instance of a class type by using the syntax:

```
<instance>.<method>(<parameters>)
```

So if I have the `String` variable named `plural` as defined above, `plural` refers to the instance of a `String`, and `plural.length()` would invoke the `length()` method on the string, and return the value of 5. I can use this expression anywhere that an integer experssion is appropriate, for instance:

```
int n = plural.length();
```

### 6.2.1 Character Positions and Accessing Individual Characters

When we wish to access an individual character of a string, we use the `charAt()` method of the `String` class. The `charAt()` method takes a single parameter that specifies the position of the desired character in the string. We begin numbering character postions of a string at index 0, so for the string `"Java"`, the 'J' is at index position 0, the 'a' at index 1, the 'v' at index 2, and the 'a' at index 3. The index of the last character of any string is equal to the length of the string minus one.

Consider the following declarations and initialization:

```
String line = "Let it be!"
int lineLength = line.length();
char firstChar = line.charAt(0);
char lastChar = line.charAt(lineLength - 1);
char middleChar = line.charAt(lineLength/2);
```

In this example, the length of the string, stored in the variable `lineLength`, is 10. In the second line, we pass an integer 0 as the requested character position in the invocation of `charAt()` and so the variable `firstChar` has the value `'L'`. In the next statement, we retrieve the last character of the string by computing the final index position of `lineLength - 1`, or 9. This yields the value `'!'` in the variable `lastChar`. Finally, we compute the position of the character about halfway through the string with the expression `lineLength/2`, which evaluates to 5, and is passed to `charAt()` to get the `'t'` assigned to the variable `middleChar`.

Note that it is an error to invoke the `charAt()` method with a parameter whose value is outside the valid range of indices for the instance string on which it is operating.

### 6.2.2 Other Useful String Methods

#### Finding the position of a substring

Another frequent operation required in programs is to search for a substring within a larger string. The `String` class provides a method for accomplishing this. Say

that we have a string variable declared and intialized as follows:

```
String quote = "All the world's a stage";
```

If we wish to find the index position of the substring `"the"` within this string, we can use the `indexOf()` method:

```
int pos = quote.indexOf("the");
```

This returns the index within the instance string of the first occurance of the specified string. In this case, the variable `pos` would be assigned the value 4. If the substring were not found in the instance string, the method returns a -1.

**Extracting a substring by position**

At other times in our programs, it may be necessary to extract a substring from a given string. The method `substring()` accomplishes this task for us. For example, if we with to extract the substring from index position 8 through index position 12 (inclusive) of the `quote` string instance, and assign the substring to a new `String` variable, we could write:

```
String sub1 = quote.substring(8,12);
```

This would result in `sub1` referring to a new string whose value is `"world"`. It is an error for the beginning index to be greater than the ending index, or for either the begiining index or ending index to be outside the range of valid indices for the string.

**Comparing two strings**

We often wish to compare two strings to see if they have the same sequence of characters. For primitive types, we can simply use the '`==`' comparison operator, but for class types, we require an operation that "looks inside" and compares the elements within. For instances of the `String` type, we can invoke the `equals()` method, that takes a single string parameter specifying the string to compare the instance string to. The method returns a `boolean` (true or false) that indicates if the instance string is identical to the parameter string or not. By returning a `boolean`, we can use this method invocation anywhere a boolean expression is appropriate, such as in the condition of an **if** statement.

Say that we have a string input from the user and want to determine if the user typed the string "stop". The following code exemplifies this common scenario:

```
Scanner input = new Scanner(System.in);
String answer = input.nextLine();
if (answer.equals("stop")) {
< stop-statements >
}
```

Since the method invocation is a boolean expression, we can apply boolean operators to it. For instance, we can change the above example to execute a block of statements whenever the user's answer is *not* `"stop"` by using the code: `if (!answer.equals("stop"))`. The parameter to the `equals` method need not be a string constant. It could be any string expression. For instance, we could compare `String string1` with `String string2` by using `string1.equals(string2)`. It is equivalent to interchange the instance string and the parameter string, so `string2.equals(string1)` will return the same result.

If, in the above example, the user typed `"Stop"` instead of `"stop"`, then the `equals()` method would return **false**, because the upper case 'S' is not the same character as the lower case 's'. This will probably result in unexpected behavior of the program. The `String` class defines a method named `equalsIgnoreCase()` to address this case. As the method name suggests, this compares an instance string with a string parameter, but ignores differences in case in the two character sequences.

A final comparison method of class `String` comes into play when we wish to determine the lexicographic ordering of two strings. Think of the lexicographic ordering of strings as their ordering by standard dictionary-style alphbetical ordering. In applications such as sorting or searching a collection of elements, we need to compare two strings and see which should occur earlier than the other in this ordering. We want to be able to tell that the string "Jones" should be ordered before "Smith" and that "Smith" should be before "Smithson", which should be before "Smithy".

The `compareTo()` method of the `String` class gives us this ordering ability. We compare an instance string, through which we invoke the method, to a string passed as a parameter. There are three possible outcomes to this comparison. If the instance string occurs *before* the parameter string in a lexicographic ordering, the method returns a negative integer. If the two strings are *equal*, the method returns zero. And if the instance string occurs *after* the instance string in the ordering, the method returns a positive integer. For the most part, we need not be concerned with the magnitude of the value returned from `compareTo`; we primarily interested in whether the result is negative, positive, or zero.

The following program illustrates the use of `equalsIgnoreCase()` and `compareTo()` in a way that allows you to experiment and see for yourself the

way the methods work.

```
1  /**
2   * The CompareStrings class illustrates the use of the compareTo
3   * and equalsIgnoreCase methods of the String class by repeatedly
4   * comparing two strings and reporting the outcome of the comparison.
5   */
6
7  import java.util.Scanner;
8
9  public class CompareStrings {
10
11   // The single method of the class, main.
12
13   public static void main(String[] args) {
14
15     String answer;     // Hold the answer from the continue question
16
17     String string1;
18     String string2;    // Hold the two strings entered by the user
19
20     int retval;        // Return value from compareTo invocation
21
22     Scanner consoleIn = new Scanner(System.in);
23
24     System.out.println("Do you wish to continue [yes/no]? ");
25     answer = consoleIn.nextLine();
26
27     while (answer.equalsIgnoreCase("yes")) {
28
29       System.out.println("Enter the first string: ");
30       string1 = consoleIn.nextLine();
31
32       System.out.println("Enter the second string: ");
33       string2 = consoleIn.nextLine();
34
35       retval = string1.compareTo(string2);
36       if (retval < 0) {
37    System.out.println("compareTo is negative: " + retval);
38           System.out.println("so string1 (the instance string) is");
39    System.out.println("before string2 (the parameter string):");
40    System.out.println("  " + string1 + " < " + string2);
41       } else if (retval > 0) {
42    System.out.println("compareTo is positive: " + retval);
43           System.out.println("so string1 (the instance string) is");
44    System.out.println("after string2 (the parameter string):");
45    System.out.println("  " + string2 + " < " + string1);
```

```
46        } else {
47     System.out.println("compareTo is zero: " + retval);
48            System.out.println("so string1 (the instance string) is the");
49     System.out.println("same as string2 (the parameter string):");
50     System.out.println("  " + string1 + " = " + string2);
51        }
52
53        System.out.println("Do you wish to continue [yes/no]? ");
54        answer = consoleIn.nextLine();
55      }
56    }
57  }
```

---

## 6.3 Writing a Loop over a String

In Chapter 5, we learned the syntax and semantics of writing loops in Java, allowing us to repeat a set of actions a number of times. When we want to perform the same set of steps for each character in a string, a loop that iterates over the sequence of characters, one character at a time, is the right tool for the job.

To illusrate, we will develop a complete program to solve a particular problem. Suppose we wish to count the number of words in a sentence that is input by the user. This will serve as our problem statement.

As we think about our solution to the problem, we note that, in a well-formed sentence, each word is followed by either a space character, or the sentence terminating punctuation character. For simplicity, let us assume that all of our input sentences end in a period, and the sentences are all simple sentences (so no commas, colons, or other punctuation in the sentence interior). This gives us our basic algorithm: examine each character in the input string. If the character is either a space or a period, increment a counter keeping track of the number of words.

The steps to solve our problem may be enumerated as follows:

1. Prompt the user for an input sentence.
2. Retrieve the input sentence from the user.
3. Begin with the word count, denoted $x$, set to zero.
4. Examine each character, denoted $c$, in the input sentence.
   (a) If the current character $c$ is either a space or a period, then increment the word count $x$.

(b) Otherwise, continue.

5. Output the final word count result, $x$, to the user.

---

```
1  /**
2   * The WordCount class retrieves an input sentence from the user and
3   * then combines a loop with String functions to examine the characters
4   * and count the words in the sentence.
5   */
6
7  import javax.swing.JOptionPane;  // Tell Java where to find
8                                   // JOptionPane
9
10 public class WordCount {
11
12   // The single method of the class, main.
13
14   public static void main(String[] args) {
15
16     int wordCount = 0;   // Running count of number of words
17
18     String sentence;     // User entered sentence
19     int sentenceLength;  // Length of user entered sentence
20
21     // Prompt the user for the sentence and retrieve input.
22
23     sentence = JOptionPane.showInputDialog(null,
24                                            "Enter a simple sentence:");
25
26     // Number of iterations of the loop is the number of
27     // characters in the sentence.
28
29     sentenceLength = sentence.length();
30
31     // Loop index: 0 <= i <= sentenceLength - 1
32
33     for (int i=0; i < sentenceLength; i++) {
34
35         // set variable to the character at the current index (i)
36
37         char currentChar = sentence.charAt(i);
38
39         // check to see if it is a space or period
40
41         if (currentChar == ' ' || currentChar == '.') {
42
43             // if so, increment the word count
```

```
44
45              wordCount = wordCount + 1;
46          }
47
48          // nothing else to do in the loop
49      }
50
51      // Display result in a friendly way
52
53      JOptionPane.showMessageDialog(null,
54                      "Sentence word count is " + wordCount);
55    }
56  }
```

# Chapter 7

# Arrays

Often we need a collection of closely related variables. For instance, suppose we wanted to perform an operation on four exam scores for the computer science course. We could declare and initialize four separate variables like this:

```
int score1 = 90;
int score2 = 78;
int score3 = 86;
int score4 = 100;
```

In Chapter 5, we saw how a loop can allow you to process a series of variables by repeating the same action over and over. However, once one iteration of the loop is over, any information in the loop variables is replaced during the next iteration of the loop; essentially, none of the specific data is stored and can be accessed after the loop terminates. Suppose instead we needed to process and store twenty exam scores. Nobody wants to declare twenty separate variables; there is a better way.

An **array** is a list or collection of variables. All the individual variables in the array must be the same type (all `int`s or all `double`s, for example) referred to as the *base type* of the array. The individual array variables are now called *array elements* and we use a single variable name to refer to the entire collection of elements. In the following example, we illustrate an array called `scores` of integer elements:

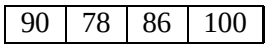



Figure 7.1: An array of exam score `int` variables

## 7.1 Basics

In Java, arrays are declared by stating the base type of the array, followed by a pair of brackets, and then an identifier or name for the array variable. Here is an example of how to declare an array of exam scores:

```
int[] scores; // declare array
scores = new int[4]; // allocate memory space
```

There are two steps needed to prepare an array variable. The first line *declares* a variable named `scores` to be an array of integers (the base type). This tells the Java compiler to associate the variable named `scores` with an integer array. The second line tells the compiler to allocates memory for four integers in the array. The keyword `new` is used to allocate new memory space for the array.

Unlike primitive types, arrays and objects require both a declaration and an allocation step which will be discussed thoroughly in Chapter 9. Both the declaration and allocation steps are necessary though most programmers will combine them into a single statement as follows:

```
int[] scores = new int[4]; // declare and allocate
```

The *size* or *length* of the array is specified in the allocation step. In the above example, the length of the `scores` array is four; there are four integer elements in this array. We can substitute any integer constant, integer variable or integer expression for the size specification.

Though the array name refers to an entire collection of integers, we access each integer element individually. Each element in an array is indexed by an integer location; in Java, the first element has index 0 instead of index 1. Syntactically, we use brackets again to denote the index of the element in the array to access. In the code below, we illustrate the basic assignment statements using arrays.

---

```
1  //  accessing integers in the array
2  public class SimpleArrayExample2
3  {
4     public static void main ( String args[] )
5     {
```

```
6         int [] scores;          // declare array
7         scores = new int[4];    // allocate memory space
8
9         scores[0] = 90;
10        scores[1] = scores[0] - 12;
11        scores[2] = 86;
12        scores[3] = 99;
13        scores[3]++;
14
15        // compute the average of the scores
16        double average = ( scores[0] + scores[1]
17                         + scores[2] + scores[3] ) / 4.0;
18     }
19  }
```

---

Notice that `scores[0]` is the first element indexed in the array (not `scores[1]`) and `scores[3]` is the last element in the array. It is an error to attempt to access an array at an index outside this range. Because it is easy to forget to start numbering at 0, a common beginning mistake is to attempt to access an array one past the last entry. Try it out to see what error message you receive.

```
int[] scores = new int[4];
scores[0] = 90; // first exam score
scores[3] = 100; // last exam score
scores[4] = 100; // out of bounds exception
```

## 7.2 Processing Arrays with Loops

It is quite common to perform the same operation to every element in an array. The for-loop is the natural construct that makes this an easy task. In this next example, we declare an array of twenty `double`s and initialize each element in the array to a random number (between 0 and 1). Remember that Java requires each variable (including each entry in an array) be initialized before use. We then compute and print the average of the numbers in the array.

---

```
1  /**
2   * Create twenty random numbers
3   * Compute and print their average
4   */
5  public class AverageTwenty
```

```
 6  {
 7     public static void main ( String args[] )
 8     {
 9
10        double numbers[] = new double[20];
11
12        for ( int i = 0; i < numbers.length; i++ )
13           numbers[i] = Math.random();
14
15        double sum = 0;
16        for ( int i = 0; i < numbers.length; i++ )
17           sum += numbers[i];
18
19        double average = sum / numbers.length;
20        System.out.println("Average is " + average + ".");
21     }
22  }
```

---

There are several new concepts here. First notice the syntax, `numbers.length` on line 12. With a `.length` appended to the variable name of an array, we can access the length (number of items) of the array; in this example, `numbers.length` has a value of 20. This is useful because if we change the size of the array in the allocation step on line 10, we do not need to change the for-loop as well; it will automatically adjust to whatever length `numbers` happens to be.

Lines 12 and 13 use a for-loop to assign a value to each `double` in the array; each array element receives a double value between 0 and 1 through a call to `Math.random()`. Lines 15-17 compute the sum of all numbers in the array. A variable named `sum` is used to accumulate the sum. Line 17 uses the short-hand `+=` operator to add the contents of each slot in the array to `sum`; this is equivalent to

```
sum = sum + numbers[i]; // same as sum += numbers[i]
```

## 7.3 Initialization

There is a short-cut notation for declaring and initializing an array in one step, useful for smaller arrays. Suppose we wanted to declare and initialize an array with the four single-digit prime numbers (2, 3, 5, and 7). Instead of using four separate assignment steps to put values into an array of four integers, we can create, allocate and initialize the array all in one step as shown in the following example.

```
1  /**
2   * Initialize an array with the four single digit primes
3   */
4  public class InitializeArray
5  {
6     public static void main ( String args[] )
7     {
8        int primes[] = {2, 3, 5, 7};
9
10       for ( int i = 0; i < primes.length; i++ )
11          System.out.println(primes[i] + " is a prime number.");
12    }
13 }
```

The Java compiler determines there are four elements in the set on the right side of Line 8, automatically allocates an array of four integers and initializes the array with the values in the set. Notice the use of the braces to hold the comma-delineated, initialization set. The for-loop on lines 10 and 11 prints each prime number.

## 7.4 Multidimensional Arrays

All the arrays discussed so far have been a sequence of data items. We call this a *one-dimensional array* because the sequence extends in one dimension (it has length). We can also create arrays that are like tables; that is, they have two dimensions (length and height). In the following code example, we create a two-dimensional array of characters to implement a Tic-Tac-Toe game:

```
1  /**
2   * A two-dimensional array to hold a game of Tic Tac Toe
3   */
4  public class TicTacToe
5  {
6     public static void main ( String args[] )
7     {
8        char board[][] = new char[3][3];
9
10       // initialize each board slot to a blank
11       for ( int i = 0; i < board.length; i++ )
```

```
12         {
13            for ( int j = 0; j < board[i].length; j++ )
14            {
15               board[i][j] = ' ';
16            }
17         }
18
19         // put an 'X' in the upper right corner
20         board[0][2] = 'X';
21      }
22   }
```

---

With two dimensional arrays, we use two pairs of brackets for declaration, allocation, and access. We typically think of the first set of brackets (the left-most pair) as specifying the rows and the second set of brackets (the right-most pair) as specifying the columns. For example,

```
int array [][] = new int[3][5]
```

declares and allocates a two dimensional array with three rows and five columns. To the Java compiler, a two dimensional array is really an array of arrays; it is an array whose base type is also an array. In the Tic-Tac-Toe example, the variable `board` is an array where each element is a character array. This concept of arrays within arrays is depicted in Figure 7.2.

$$\texttt{board} = \begin{bmatrix} \texttt{board[0]} \rightarrow \begin{bmatrix} \texttt{board[0][0]} & \texttt{board[0][1]} & \texttt{board[0][2]} \end{bmatrix} \\ \texttt{board[1]} \rightarrow \begin{bmatrix} \texttt{board[1][0]} & \texttt{board[1][1]} & \texttt{board[1][2]} \end{bmatrix} \\ \texttt{board[2]} \rightarrow \begin{bmatrix} \texttt{board[2][0]} & \texttt{board[2][1]} & \texttt{board[2][2]} \end{bmatrix} \end{bmatrix}$$

Figure 7.2: Structure of a Two-dimensional Array

Here `board` refers to the outer-most array. Each element in `board` is an array and is depicted in Figure 7.2 as a row. Notice `board.length` provides the number of rows in the `board` array and that `board[0]` accesses the first element of `board` (which is the array in the top row) while `board[2]` accesses the last element of `board` (which is the array in the bottom row).

We access the length or a particular element of an inner array by the following syntax: `board[0].length` gives the length of the first inner array while `board[1][2]` refers to the middle inner array (`board[1]`) and accesses the last element from that middle row.

On lines 11-17, nested for-loops (one loop inside the other) initialize each character in the array to a blank space. The outer for-loop (with $i$ as an index counter) goes row by row while the inner for-loop (with $j$ as a loop counter) processes column by column. You might wish to trace through this code examine by executing each statement and noting the values of $i$ and $j$ as you fill blanks into the table (two-dimensional array). Just as single for-loops are the standard way to operate on each element of a one-dimensional array, nested for-loops are the obvious way to perform an operation on each element of a two-dimensional array.

The last statement on line 20 makes a move for Player 1 in this game. The spot in the first row (row = 0) and last column (column = 2) is marked with an `'X'`. This assignment statement puts an `'X'` in the upper right corner as Player 1's first move of the game. Recall that the array basetype is `char` and that `'X'` is a character constant.

You can also create arrays of three or more dimensions though these are rarer. Can you guess the syntax for declaring and accessing such an array?

## 7.5 An Application Example

Consider the following problem:

> Construct a program that searches through a sequence of integers to find the smallest number.

**Design**

To store the sequence of integers, we opt for an an array. Since the problem statement (above) did not specify exactly how many integers are to be in the sequence, we have a few choices. First we could declare a constant at the beginning of our program and use this value throughout; this makes it easy to change the number integers quickly and in only one place in our program. Alternatively, at the start of the program we could prompt the user for the number of integers. This second option allows the program to adapt to the user without having to recompile. Let's

opt for this second choice here.

After we receive the number of integers from the user and declare/allocate our array, we must fill it with data. In this program, we have the user enter the sequence of integers using the dialog boxes of a previous chapter. Prompting the user for the numbers is convenient for small arrays and for quickly testing our program, but to handle larger data sets, we would probably want to read array values from an input file or create them automatically (randomly, for example).

Finding the smallest value in an array is a little trickier than might first appear. At this point, we encourage you to see if you can sketch an algorithm to solve this task yourself. It is a worthwhile programming exercise. Read on when you are ready to see how we solve the problem.

We use the variable `smallestSpot` to keep track of the location of the smallest item in the array. As we scan the array from start to end, we will compare each new array element with what we currently believe to be the smallest location; if we find a new item smaller, then we will update the smallest spot. It is critical to initialize `smallestSpot` to the first index (location 0) in the array before starting the loop. Do you see why this step is necessary?

### 7.5.1 Edit

We enter the following program into Dr. Java and save it in a file called `SmallArray.java`.

```
1  /**
2   * An application to find the smallest value in the array
3   */
4  import javax.swing.JOptionPane;
5
6  public class SmallArray
7  {
8     public static void main ( String args[] )
9     {
10
11       String answer = JOptionPane.showInputDialog(
12          null,
13          "How many integers in your array?");
14       int num = Integer.parseInt(answer);
15
16       // create an array of NUM integers
17       int array[] = new int[num];
```

```
18
19          // fill the array with values from the user
20          for ( int i = 0; i < array.length; i++ )
21          {
22             answer = JOptionPane.showInputDialog(
23                null,
24                "Enter integer " + (i+1) + ":"
25                );
26             array[i] = Integer.parseInt(answer);
27          }
28
29          // find the location of the smallest integer in the array
30          int smallestSpot = 0;  // assume the smallest int is in the first spot
31          for ( int i = 0; i < array.length; i++ )
32          {
33             if ( array[i] < array[smallestSpot] )
34                smallestSpot = i;
35          }
36
37          // report the smallest value
38          JOptionPane.showMessageDialog(null,
39             "Smallest int is " + array[smallestSpot]);
40       }
41    }
```

---

### 7.5.2 Compile

If you didn't type the above program correctly, you may need to fix your compile errors.

### 7.5.3 Execute

Test the program several times with arrays of different sizes and values, and with different locations for the smallest item. Developing thorough test cases is both a science and an art form that helps eliminate bugs from the code.

## 7.6 Exercises

**Exercise 7.1** *Declare (only declare) an array of fifteen characters named* `word`*.*

**Exercise 7.2** *Allocate memory for the* `word` *array in Exercise 7.1.*

**Exercise 7.3** *Declare and allocate an array of 100 integers for holding the homeworks grades of a very large class (assume one homework grade for each of the 100 students – a one dimensional array).*

**Exercise 7.4** *Declare and allocate an array to hold all the lower case vowels.*

**Exercise 7.5** *Declare, allocate and initialize an array to hold all the lower case vowels. Do this in at least two different ways.*

**Exercise 7.6** *Declare, allocate and initialize an array to hold the first four letters of the lower case alphabet. Do this in at least three different ways. Hint: can you use a loop to initialize the array? Is it possible to use a* `char` *type for a loop counter?.*

**Exercise 7.7** *Establish an array to hold the integers 1 through 10. Use a loop to initialize the array.*

**Exercise 7.8** *Denison has 2000 students who will each take four courses this term. Allocate an appropriate array to hold the final grades for each student in each course. Hint: use a two dimensional array here.*

**Exercise 7.9** *Assume the array of grades in the previous question has been allocated and filled-in. Write a nested loop that will compute the GPA for each student during this term. You can assume the grades are all A, B, C, D, or F. Store the GPAs in another array that you create, declare, and allocate.*

**Exercise 7.10** *Again there are 2000 Denison students. Now allocate an array that will hold the four course grades for each student during each of their eight semesters. This is a three-dimensional array. As a challenge, see if you can write a loop to compute the overall GPA for each student (assuming the grades for all eight semesters are present).*

# Chapter 8

# Methods

A *method* gathers together a set of program steps that define an action in a program. We have already used a number of built-in methods that are part of the Java API. The `System.out.println()` method prints messages to the console. The `Math.sqrt()` method computes the square root of a number. In addition to the numerous built-in methods, we canconstruct our own that perform tasks specific to our program. In fact, we have already created a method called `main` in each of our application programs.

In this chapter we learn the basics of designing and using our own methods that will enable us to better manage the growing complexity of our programs. As we shall discuss, methods are important for a number of reasons. Conceptually, they allow us to break apart long and complex programs into more manageable pieces. Methods allow us to better re-use our own code and share our code with other programmers; this reduces the amount of effort required to build new programs. Proper use of methods is important for isolating errors in programs and promotes faster debugging. We'll discuss the full implications of these factors after we have learned the basics of methods.

## 8.1 Basics

We start with the syntax for defining a method:

```
1  <modifiers> <return-type> <method-name> ( <parameter-list> )
2  {
3     <statements for method body>
4     ...
```

```
5     <return statement>
6  }
```

---

Here is an example of a method called `addTwoNumbers()` in our own program.

---

```
1  /**
2   * This program illustrates a basic method.
3   */
4  public class MethodsExample1
5  {
6     public static void main ( String args[] )
7     {
8        int x = 5;
9        int y = 3;
10       int z = addTwoNumbers(x,y);
11       System.out.println(x + " + " + y + " = " + z);
12    }
13    /**
14     * A method to add two numbers and return the sum.
15     */
16    public static int addTwoNumbers ( int num1, int num2 )
17    {
18       int  sum;
19       sum = num1 + num2;
20       return sum;
21    }
22 }
23
```

---

Let us examine each part of the method separately.

- *Method placement*
  Methods are created and listed inside the class definition. If the class is an application, the `main` method is typically listed first. In `MethodsExample1.java`, we have also created a method called `addTwoNumbers()` which is listed inside the class but after the `main()` method.

- *Method name*
  We name each method so that we can call upon the method to perform the

desired action. Since method names are identifiers, the syntax rules for them are the same as for identifiers used in other parts of our programs such as for variable and class/program names. It is conventional to choose verbs for our method names and to capitalize the first letter in each word of our method name except for the first word. The `addTwoNumbers()` identifier follows this convention.

- *Return type*
  Immediately preceding the method name on the first line, is the return type. The return type can be any valid primitive Java type, such as `int` or `double`, or any Java object (more about these in a later chapter). We may also use the keyword `void` to indicate no return type.

  For example, `println()` is a method that does not return any type. Its job is merely to print a message to the console. In contrast, `sqrt()` is a method that returns an answer (specifically, the square root of a number). Four our `addTwoNumbers()` method, we specify `int` as the return type to indicate that our method returns an integer answer.

  Methods can return only one answer though you can return arrays and objects which hold more than one single data item.

- *Modifiers*
  You'll notice that the `addTwoNumbers()` method starts on line 16 with the keywords `public static`. These modifiers precede the return type and affect how the method is accessed and how it behaves. In Chapter 9 on objects, we'll examine the meaning of the modifiers, but for now we'll simply include them in the methods we write.

- *Parameters*
  The last item on the first line of a method definition is the *parameter list*. The parameters are enclosed in a set of parentheses and consist of matched pairs of types and variables. The parameters act much like variables inside the method but are also a conduit for information coming in to the method from the outside world. We'll further discuss the details of parameters later in this chapter.

- *Method body*
  The method body is a series of statements that perform the work of the method and is enclosed in a pair of start and end braces. Methods that return values (the return type is other than `void`) contain a special `return` statement that indicates the exact value to be returned; the variable or expression in the return statement must match the return-type specified in the first line of

the method declaration. Methods of type `void` may have return statements where no return value is specified; commonly, these methods omit the return and the Java compiler automatically inserts a return as the last statement in the body. Notice our `main()` method (of type void) does not contain a return statement.

## 8.2 Invocation and Execution Order

Method calls alter the order in which statements are executed. The program starts by executing the first statement inside the `main` method and then continues executing statements, one at a time, in the order they are listed. However, on line 10 of the `MethodExample1.java` program, we encounter the call to the `addTwoNumbers()` method. The execution of statements in `main()` will suspend at this point. The program then "jumps" to the first line in the `addTwoNumbers()` method; it will execute statements in the method body until it reaches the end or encounters a `return` statement. Once execution of the `addTwoNumbers()` method is complete, the program will then return back to the `main()` method at line 11 and resume executing the remaining statements inside `main()`.

## 8.3 Parameters and Passing Information

Parameters are the information passed in to the method. We distinguish between two different kinds of parameters, formal and actual parameters. *Formal parameters* are the variables in the method; on line 16, `num1` and `num2` are the two formal parameters for this method. *Actual parameters* are the specific values being passed to the method at the method call. These are the `x` and `y` parameters on line 10.

Actual parameters are matched to the formal parameters in the method's parameter list; it is an error if the types of these matchings are not consistent. Values are copied from the actual parameters and passed to the formal parameters. Thus, the formal parameter `num1` receives a value from actual parameter `x` and formal parameter `num2` receives a value from actual parameter `y`.

In order to understand the connection between formal and actual parameters, we'll need to outline how variables are stored in memory. Consider the following snippet of code:

```
int num = 1;
int scores[] = { 90, 85, 100};
```

For primitive data types such as `int`, java allocates a single location in memory. The variable name, `num` in this example, is associated with this memory

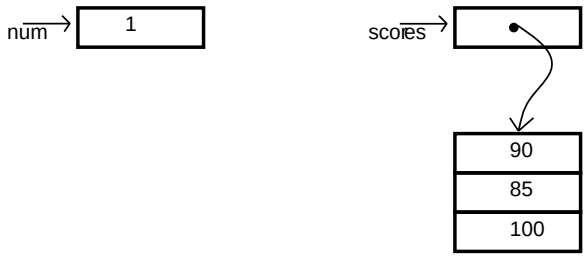



Figure 8.1: Memory Allocation for Variables

location. The actual value of the variable is stored here. In Figure 8.1, we see how this segment in memory is reserved, how the variable name is associated with this location, and how the value is stored here.

Objects and arrays, called *reference-types*, are treated differently than primitive types. Because objects and arrays typically contain more than one data element, they require more than a single memory location. In our example, the array `scores` is still allocated an initial single memory location. However, instead of storing the whole array here (it won't fit), we store a reference to separate, larger segment of memory which contains all the data elements for the array. This also explains why arrays (and objects) require two steps. During declaration the array name is associated with the single reference location. But during allocation, the reference points to a larger segment of memory which is reserved for the array. This two-step concept is also illustrated in Figure 8.1.

Now that we have a basic understanding of how Java reserves memory for primitive variables and arrays, we can explore how Java associates formal and actual parameters. Java uses a concept called pass-by-value to exchange information between formal and actual parameters. In *pass-by-value*, only the value of an actual parameter is copied to the formal parameter. The formal parameter in the method is a separate variable. The programmer is free to change the value of the formal parameter inside the method, but any changes here do not causes changes to the actual parameter in the method call.

We turn to another example program to help illustrate the subtle effects of pass-by-value; see `MethodsExample2.java`. In this program, the method named `doNothing()` is an example of how primitive types are affected with pass-by-value. On line 8 of the main program, `num` is assigned the value of 1 as is confirmed by the first `println()` statement. Then `doNothing()` is invoked and variable `num`, used as the actual parameter, is passed in by value. The `doNothing()` method receives `num`'s initial value and assigns it to the formal parameter named

`x`. Inside of `doNothing()`, we print the value of `x` (it is 1), then change the value to 0, and print again (it is now 0). After `doNothing()` returns, we print the value of `num` again in the main program. The value of `num` is still 1 showing that changes to the formal parameter inside `doNothing()` do not cause changes to the actual parameter `num`.

---

```
1  /**
2   * This program illustrates parameter passing in methods
3   */
4  public class MethodsExample2
5  {
6     public static void main ( String args[] )
7     {
8        int num = 1;
9
10       System.out.println("num = " + num);
11       doNothing(num);
12       System.out.println("num = " + num);
13
14       int scores[] = {90, 85, 100};
15
16       System.out.println("scores[0] = " + scores[0]);
17       doSomething(scores);
18       System.out.println("scores[0] = " + scores[0]);
19    }
20    /**
21     * Illustrates pass-by-value
22     */
23    public static void doNothing ( int x )
24    {
25       System.out.println("x = " + x);
26       x = 0;
27       System.out.println("x = " + x);
28    }
29    /**
30     * Illustrates pass-by-reference
31     */
32    public static void doSomething ( int list[] )
33    {
34       System.out.println("list[0] = " + list[0]);
35       list[0] = 0;
36       System.out.println("list[0] = " + list[0]);
37    }
38 }
39
```

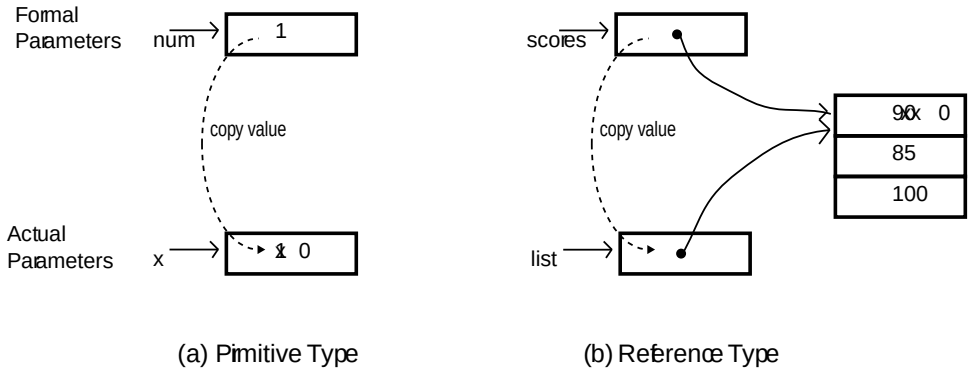



Figure 8.2: Nuances of Pass-by-value

---

This concept of having separate memory locations for the formal and actual parameters is illustrated in Figure 8.2(a). We see how the value of actual parameter, `num`, is copied to a separate memory location for formal parameter, `x`. Changes made to `x` inside the `doNothing()` method do not affect the value for `num` because `x` has its own separate memory location.

However, this same idea of pass-by-value, affects arrays and objects quite differently. In Figure 8.2(b), we see the reference associated with the formal array parameter `scores`. This reference is copied via pass-by-value to the actual parameter `list`. But notice, now there are two references pointing to the same chunk of memory holding the array data! Because they are the same array, any changes that are made to this data through the actual parameter of `list` are reflected back to the array of `scores`. In reference types, pass-by-value creates an alias (another reference) for the same piece of memory; it is akin to having two separate names for the same data.

This concept is supported in `MethodsExample2.java` by the `doSomething()` method. It accepts an array as input and changes the value of the first element. This change is retained back in the main program. Be sure to enter this program, execute it, and watch what is printed to the screen.

## 8.4 Scope

Now that we have more than one method in our programs, it is possible that we may wish to use the same variable name in different places. Java provides a rule

about how to arbitrate naming conflicts among variables. We must also understand where in a program we can use different variables that we create. All of these issues fall under the heading of variable scope. The *scope* of a variable is the part of a Java program in which the variable can be accessed.

Java provides two different kinds of scope: class and local. A variable that has *class scope* is declared within a class but outside any method; these are often called *class variables* and can be used anywhere in the class (in any method within the class). A variable that has *local scope* is declared inside a method and can only be used in that method. Since Java variables must be declared before they can be used, local scope variables can only be used in statements that follow a variable declaration inside that same method.

---

```
1  /**
2   * This program illustrates scope rules
3   */
4
5  public class Scope
6  {
7     static int   x = 1;
8     static int   y = 2;
9
10    public static void main ( String args[] )
11    {
12       int x = 3;
13
14       System.out.println("x = " + x);
15       System.out.println("y = " + y);
16
17       doNothing(y);
18    }
19    /**
20     * Illustrates scope rules
21     */
22    public static void doNothing ( int x )
23    {
24       int z = 5;
25
26       for ( int i = 1; i <= 3; i++ )
27       {
28          double root = Math.sqrt(i);
29          System.out.println(root);
30       }
31
32       System.out.println("x = " + x);
```

```
33         System.out.println("y = " + y);
34         System.out.println("z = " + z);
35      }
36   }
37
```

---

`Scope.java` is a Java program that would probably not be written other than to illustrate various scope rules. Near the top of the program on lines 6 and 7, variables $x$ and $y$ are declared outside any method; these are called *class variables* since they belongs to the class but not to a specific method within the class. In Chapter 9 we shall learn more about class variables. These class variables can be used throughout the entire java program – they have class scope.

Lines 9-17 contain the `main()` method. Line 11 declares a local variable named $x$. Variable $x$ can only be used within the body of `main()`, specifically on lines 12-17. In `main` we have a name conflict. There is a class variable $x$ and also a local variable $x$. Which variable will be printed on line 13? In this case, the local variable *overrides* or *hides* the class variable; the local variable ($x = 3$) will be printed on line 13 while the class variable $y$ is printed on line 14.

In the method `doNothing()` we have two more local variables. One of them, $z$, is easy to identify. The other is a parameter (also named $x$). Formal parameters act just like local variables inside a method except that they are initialized with values passed in from the method call. Again, local variable $x$ hides the class variable $x$.

Just to make things more interesting, notice the method call to `doNothing()` on line 16 in `main` passes the value of instance variable $y$. This value ($y = 2$) is assigned to the parameter/value $x$ inside the `doNothing` method.

In `doNothing()`, we point out one more peculiarity of scope. The for-loop on lines 26-30 contain two more variable declarations: the loop counter `i` and a data variable `root`. In this case, the for-loop creates an ”inner block” of code within the body of the `doNothing()` method. These two local varaibles, `i` and `root`, have scope that is limited only to the for-loop block of lines 26-30. It would be an error to attempt to use them outside this block. In fact, all local variables have scope restricted to the block in which they are defined whether this block is a whole method, or a smaller block created by a loop or a decision statement.

Normally, a programmer would not write a confusing program such as this. A more judicious use of names will help anyone reading your program to follow the intent and execution more easily. We merely provide this unusual example to illustrate the peculiarities of scoping rules.

## 8.5 Method Overloading

It is possible to have more than one method of the same name. In order to do this, the parameter list must be different (specifically different types in different orders). This way the java compiler will know which method to associate with each method call. Having different methods of the same name is known as *overloading* the method.

At first, you might think naming separate methods the same is poor programming practice because it adds confusion. However, there is a plausible situation in which using the same name is appropriate. Consider that you want to perform the same operation but on different data types. In `Overload.java`, we have two separate `addTwoNumbers` methods, one for integers and one for doubles.

```
 1  /**
 2   * This program illustrates method overloading.
 3   */
 4  public class Overload
 5  {
 6     public static void main ( String args[] )
 7     {
 8        int x = addTwoNumbers(2,3);
 9        System.out.println("2 + 3 = " + x);
10
11        double y = addTwoNumbers(2.5,3.1);
12        System.out.println("2.5 + 3.1 = " + y);
13     }
14     /**
15      * A method to add two integers and return the sum.
16      */
17     public static int addTwoNumbers ( int num1, int num2 )
18     {
19        return num1 + num2;
20     }
21     /**
22      * A method to add two doubles and return the sum.
23      */
24     public static double addTwoNumbers ( double num1, double num2 )
25     {
26        return num1 + num2;
27     }
28  }
29
```

The method call on line 8 has two integers as parameters. Recognizing this, the Java compiler associates this method call with the `addTwoNumbers` method on lines 17-20. Similarly, the method call on line 11 (with two real numbers) invokes the `addTwoNumbers` method on lines 24-27. This is a better design than having a method named `addTwoIntegers` and another method named `addTwoDoubles`. Since both methods accomplish the addition, it is easier for the programmer using our methods to simply invoke `addTwoNumbers`.

## 8.6 Methods and Software Engineering Principles

The addition of methods allows programmers to adopt some sound software engineering techniques.

- *Modula design*
  We want to avoid having methods (especially `main`) become too large and cumbersome. This makes it difficult to read and difficult to debug. The idea of *modular design* is to break up large tasks into smaller, more manageable units that each become a method, and then call each method to accomplish the task.

- *Code Reuse*
  Using methods makes it easier to re-use code we have already written. If a common task is written as a method, we can easily copy that method into other programs that we write. Later, when we study objects, we will learn even better ways to organize our code to make it more flexible and usable to other software that we or others create.

- *Isolation*
  We can view a method as a black box that accepts input and produces output. If we make methods that are small and compact, we can test our methods thoroughly to be sure they implement the intended operation. Once this is done, we can then use this method in other parts of our program freely without having to worry about debugging the method. If there is an error, methods make it easier to isolate and repair the bug.

Let us illustrate the principles in the following program design.

> Julie is taking a chemistry class and Megan is taking a biology class. They each have received several midterm exam grades. Based on these grades, determine who has the higher science class GPA?

Shown below in `GPA1.java` is the complete program without methods. Let us now design a solution that uses methods and compare the two.

---

```
1  /**
2   * This program does not use methods to compute GPA's
3   */
4
5  import java.util.Scanner;
6
7  public class GPA1
8  {
9     public static void main ( String args[] )
10    {
11        Scanner consoleIn = new Scanner(System.in);
12       // create array to hold Julie's scores
13       System.out.print("Enter number of midterms for Julie: ");
14       int numJulieExams = consoleIn.nextInt();
15       double julieScores[] = new double[numJulieExams];
16
17       // create array to hold Megan's scores
18       System.out.print("Enter number of midterms for Megan: ");
19       int numMeganExams = consoleIn.nextInt();
20       double meganScores[] = new double[numMeganExams];
21
22       // read Julie's scores
23       for ( int i = 0; i < julieScores.length; i++ )
24       {
25          System.out.print("Enter score " + (i+1) + " for Julie: ");
26          julieScores[i] = consoleIn.nextDouble();
27       }
28
29       // read Megan's scores
30       for ( int i = 0; i < meganScores.length; i++ )
31       {
32          System.out.print("Enter score " + (i+1) + " for Megan: ");
33          meganScores[i] = consoleIn.nextDouble();
34       }
35
36       // compute Julie's GPA (average midterm)
37       double julieAverage = 0.0;
38       for ( int i = 0; i < julieScores.length; i++ )
39       {
40          julieAverage += julieScores[i];
41       }
42       julieAverage = julieAverage/julieScores.length;
43
44       // compute Megan's GPA (average midterm)
```

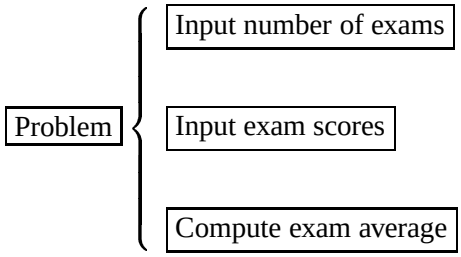
```
45         double meganAverage = 0.0;
46         for ( int i = 0; i < meganScores.length; i++ )
47         {
48            meganAverage += meganScores[i];
49         }
50         meganAverage = meganAverage/meganScores.length;
51
52         if ( julieAverage > meganAverage)
53            System.out.println("Julie has a better GPA.");
54         else if ( meganAverage > julieAverage)
55            System.out.println("Megan has a better GPA.");
56         else
57            System.out.println("They have the same GPA.");
58      }
59   }
```

---

### 8.6.1 Design

The `main()` method in `GPA1.java` is a bit too long. It becomes difficult to read and contains a number of different tasks. Throughout `main()`, we also do the essentially same task repeatedly to different data elements. It would be better to reduce the clutter by moving some code to methods.

Our first activity is to engage in modular design to think about how to break up the larger problem statement into subproblems. It seems that for each student, we must prompt for the number of midterm exams, enter and record the midterms, and then compute their average. Finally, after completing these tasks for each student, we will compare their GPAs and print a message. Each of these subtasks is about the appropriate size for a method so we create a method for each one.




By moving each task to a method, we reduce the repetitive nature of the code in the main program. In `GPA1.java`, we are repeating the same pieces of code for each student. Methods reduce the amount of code by moving oft repeated sections to a method body. Thus there is only one place in the program that contains each section; multiple method calls then invoke this code as many times as needed.

### 8.6.2 Edit, Compile, Execute

Below is the final program using methods. In `getArraySize`, we decide to pass in a `String` containing the student's name. Thus the method can be used to prompt for both students' exam numbers. Notice we use the pass-by-value feature that allows methods to change values of arrays in the `readArrayValues()` method. This method changes the values in the original array by reading values typed by the user.

---

```
 1  /**
 2   * This program uses methods to compute GPA's
 3   */
 4
 5  import java.util.Scanner;
 6
 7  public class GPA2
 8  {
 9     public static void main ( String args[] )
10     {
11        int size;
12
13        size = getArraySize("Julie");
14        double julieScores[] = new double[size];
15
16        size = getArraySize("Megan");
17        double meganScores[] = new double[size];
18
19
20        // read Julie's scores
21        System.out.println("Enter Julie's scores.");
22        readArrayValues(julieScores);
23
24        // read Megan's scores
25        System.out.println("Enter Megan's scores.");
26        readArrayValues(meganScores);
27
28        // compute Julie's GPA (average midterm)
29        double julieAverage = computeAverage(julieScores);
30
31        // compute Megan's GPA (average midterm)
32        double meganAverage = computeAverage(meganScores);
33
34        // compare GPAs
35        if ( julieAverage > meganAverage)
36           System.out.println("Julie has a better GPA.");
37        else if ( meganAverage > julieAverage)
```

```
38            System.out.println("Megan has a better GPA.");
39         else
40            System.out.println("They have the same GPA.");
41      }
42
43      /**************************************************
44       * Method to prompt for array size
45       **************************************************/
46      public static int  getArraySize ( String name )
47      {
48         Scanner consoleIn = new Scanner(System.in);
49         // create array to hold name's scores
50         System.out.print("Enter number of midterms for " + name + ": ");
51         int num = consoleIn.nextInt();
52         return num;
53      }
54
55      /**************************************************
56       * Method to read values in array
57       **************************************************/
58      public static void readArrayValues ( double[] array )
59      {
60         Scanner consoleIn = new Scanner(System.in);
61         for ( int i = 0; i < array.length; i++ )
62         {
63            array[i] = consoleIn.nextDouble();
64         }
65      }
66
67      /**************************************************
68       * Method to compute average of values in array
69       **************************************************/
70      public static double computeAverage ( double[] array )
71      {
72         double average = 0;
73         for ( int i = 0; i < array.length; i++ )
74         {
75            average += array[i];
76         }
77         return average / array.length;
78      }
79
80   }
```

## 8.7 Exercises

**Exercise 8.1** *Write a method called* `max()` *that accepts two integers as input and returns the value of the larger one.*

**Exercise 8.2** *Override the method of the previous example to work with a pair of* `double` *inputs and return a double value.*

**Exercise 8.3** *Write a method called* `wordCount()` *that accepts a* `String` *as input and counts the number of words in the string. Words are groups of character(s) separated by white space (spaces, returns, tabs) or punctuation.*

**Exercise 8.4** *Name and describe the two different types of parameters. Give a brief program and show/label each type of parameter within your program.*

**Exercise 8.5** *Name and describe the technique that Java uses to associate formal and actual parameters. Describe how this concept applies to primitive types and reference types differently; use both a program and also an picture of memory to support your description.*

**Exercise 8.6** *Is it an error to name a local variable using the same identifier as a method name elsewhere in the class? Try it out to see what happens. Even if it is legal, why should this not ever happen in one of your programs?*

**Exercise 8.7** *Give at least three reasons why it is important to write a comment at the top of each method that describes what the method does. Keep in mind that other people might be reading your code at some point.*

**Exercise 8.8** *Using the discussion of how memory is allocated for reference types, describe why arrays have both a declaration and an allocation step. What happens to the memory for an array during each step?*

**Exercise 8.9** *Write a program that allows a user to play a game of Tic-Tac-Toe against the computer. Use modular design to break the program down into simple tasks. Then write a method for each task. Have the computer pick an empty square at random for its moves.*

**Exercise 8.10** *Write a program that will spell check text documents. You should be able to obtain a text dictionary on the internet or you can create your own mini-dictionary to test your program. Prompt the user of your program to enter filenames for both a text document and a dictionary. Then look up each word in the text document to see if it is in the dictionary or not. Print any words that are not found.*

# Chapter 9

# Objects

All data stored in a java program is either a primitive data type or an object type. We have already encountered several primitive data types including `int` and `double`. The object types are further divided into array types and user-defined types. In Chapter 7 we encountered the array types. In this chapter we take a closer look at the user-defined types. Figure 9.1 depicts the relationship of the data storage hierarchy in java programs.

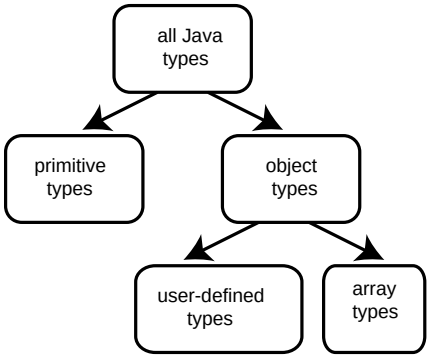



Figure 9.1: Java Data

There is often some confusion with the use of terminology in object-oriented programming. Rightly so, because some terms have different formal and informal meanings. Java uses the term *class* to refer to a blueprint, or plan, for an object. The class describes how an object is to be constructed and what features/methods it implements. If you were in the house construction business, a class is like a blueprint for the house to be constructed. As we'll see shortly, the keyword `class`

is used in programs to syntactically declare/create new types.

Formally, the term *object* refers to any instantiation of a class. Using our house construction business analogy, an object is a physical house that has been constructed according to the specified blueprint. Notice that with one blueprint (class) you can build many houses (objects). This relationship is depicted in Figure 9.2. Objects have lifespans; they come into existence, serve a purpose, and then are disposed of once their function is accomplished. Classes are abstractions; they are ideas that are ”created” and then live for all time. For example ”democracy” really isn’t a physical entity; it is an idea about how to conduct a government. The idea of democracy will be around forever even if there are no current implementations of democracies in existence.

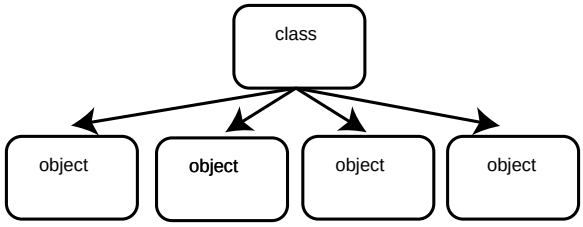



Figure 9.2: Class (blueprint) and Object (house)

Informally, the term *object* is sometimes used in place of class. That is, object denotes not only the physical implementation of a class, but also the abstract design (ie class) itself. We’ll use the term object to refer only to the physical implementation of a class but beware of this other common and casual use of the word in other literature or when conversing with programmers.

Finally, the term *abstract data type* is sometimes used as a synonym for class. Other computer scientists use abstract data type to refer to the abstract concept while a *class* describes the detailed implementation of that abstraction (and objects are physical instantiations of the class). Think of ”Victorian style” as an abstract data type of our house construction project; we can have different blueprints (classes) detailing Victorian houses and how they are to be constructed.

In summary, classes describe a new data type while objects are physical implementations of that data type. These features allow the programmer to create new data types that are useful to the specific program being written. They also extend the language by allowing programmers to create libraries of common classes and share them with others, thereby reducing redundant programming efforts. With the right design philosophy, objects and object-oriented programming facilitate the creation of very large programs.

In this chapter we examine the basics of creating our own classes.

## 9.1 The Philosophy of Objects

Objects provide three key features: encapsulation, data hiding and inheritance. These three features allow the programmer to extend the Java language by creating new data types.

We use the automobile as an analogy to develop some of these ideas. People with a small amount of training and a proper license are able to drive just about any kind of automobile even though each car may differ greatly in its design and structure.

*Encapsulation* refers to the logical grouping of a data type along with the operations on that data type. In a non-object-oriented programming language, say C for example, the programmer primarily builds a structure to hold some particular data; then routines are designed to operate on those structures. In Java, the view is more wholistic. The data item and its operations are viewed as a natural whole.

In terms of our automobile, encapsulation refers to idea that you view the automobile as a unified concept. For example, you don't go down to the local Ford dealership to buy some tires, an engine, a frame, a transmission, some assorted nuts and bolts, and then take everything home to assemble your own home-made automobile. Instead, you buy the whole car, already fully assembled and ready to go.

This idea of encapsulation makes more sense if you consider that two kinds of people are usually involved in an object. The class designer is like the car manufacturer; they must know all the nuts-and-bolts details of their car. The class user is like a driver. They don't usually need to know much about how the car actually works; they only need to know the basics of driving and they are ready to use just about any automobile. Though you will be creating and using your own objects for now, try to think of yourself in these two roles; object oriented programming makes more sense when seen in this light.

Closely related to encapsulation is the idea of *data-hiding*. A class typically has two perspectives: the class's public view and its implementation. The public view, often called its *abstract data type*, is intended for users of the class. It describes what the class does and how the user can interface to the class. The class's implementation, however, is typically hidden from view. Only the class's programmer/creator sees the implementation. The implementation contains many details that are not relevant to the use of the class and thus are not part of the public view; as long as the object works as advertised in its abstract data type, the user should not need to know how the object actually accomplishes all its tasks. Re-

call the previous distinction between the abstract data type (public view) and class (private implementation detailed in a blueprint).

When you drive an automobile, you are using the car's public interface. You know there is an ignition switch, a gear shifting mechanism, a steering wheel, a brake pedal and accelerator, possibly a clutch, and an instrument panel. All cars have these things and they all work pretty much the same way. However, the internal workings of a car vary greatly. Your Jeep might have an inline 6-cylinder engine while your Mazda sports car uses a rotary engine. These are details that you don't need to know in order to operate the car. Only your car's designer, fabricator, and mechanic really need to be aware of the car's internal workings. These internal workings – the implementation of its public interface – are hidden from the user's view. This is what makes driving a car a relatively easy task for all kinds of people, especially those with limited mechanical experience.

Lastly we discuss the notion of *inheritance*. Inheritance is the idea that instead of creating a separate class for two closely related data types, we can often combine the design and construction of classes by sharing their resources.

Consider that there are many different kinds of cars: sports cars, sedans, wagons, sport-utility vehicles, etc. They all share common properties (see list above) but they also have features in common within each category (sports cars are typically low to the ground and fast). Instead of creating completely separate classes for each type of car, we first might create a generic class called car that contains those things in common (steering wheels, engine, ...). Then we can create separate "extension classes" for each type of car that inherit all the basic properties of a car. In this way, we are not duplicating the same common attributes everywhere. Inheritance is one feature of object-oriented programming that will not be addressed further in this text.

## 9.2 Class Basicis

In this section we create our first class. We'll look at the java specific syntax of how classes are declared while we keep in mind the terminology and class design philosophy discussed previously. We use a deck of cards for an illustrative example.

### 9.2.1 The Abstract Data Type

Before we write any code or discuss class syntax, we first need to think wholistically about the deck of cards as an object. Take a moment and jot down a few ideas on scratch paper. Try to think about how you interact with a deck of cards.

What important features do decks have? In what ways do you use them? Are there some uses that are generic to all decks? Are there others that are specific to one particular use (say dealing seven cards for Go Fish)? Return back to this discussion when you have spent a few minutes building your own list.

Our list consist of the following:

- A deck has 52 cards.
- There are four suits (clubs, diamonds, hearts, and spades).
- There are thirteen cards in each suit (2, 3, ..., 10, Jack, Queen, King, and Ace).
- We often want to deal cards (extract one card at a time) from the deck.
- We'll need to shuffle the cards in a random order.
- We may need to recollect all the cards and start over by placing them all in the deck.

Our list is fairly short. Every time I think about adding something else, I hesitate because I want my list to contain only those things shared by all decks and common uses. Specialty things (like jokers or like the ability to add cards back to the deck) can be accomplished later by extending the class via inheritance. Good object designers (ie class designers) tend to be minimalists; they include only those things which are necessary and inherent in the object.

For now, you can imagine our list above as the beginnings of an abstract data type (ADT) for the deck of cards. Think of the ADT as a contract with the class's user: these are the things I promise my deck class will accomplish. At this point, an experienced java programmer would probably want to write a fully specified ADT document; this would help guide them in the creation of their object, especially if multiple people are involved in the effort. Since this is our first spin with objects, we'll forgo this step and start the class design in java-specific syntax.

### 9.2.2 Syntax Rules for Classes

Every java class has a class name. Though any valid java identifier will do, the convention is to use a capital for the first letter in each word and to make the class name a noun. Good examples are `VictorianHouse` and `SportsCar`. We'll use `CardDeck` for our class name.

Every java class must be created in a separate text file. The name of the file must be identical to the class name except that `.java` is appended to the file-name. We open a text editor (Dr. Java will do) and create a new file with the name `CardDeck.java` for our deck class.

As shown in the code segment below, each java class begins with the keyword `class` followed by the class name. A pair of braces then enclose the contents of the class. Often the keyword modifier `public` precedes the class definition as is shown below.

```
1  public class Deck
2  {
3    // instance variables are listed here.
4
5    // methods are listed here.
6
7  } // end of Deck class
```

Classes typically contain two things: data structures and methods. The data structures are used to store the data for the object. These can consist of primitive typed variables, arrays or more user defined types. They are sometimes called *instance variables* for the class. The second major component in each class is a list of methods that provide the usable features of the class. It is typical to list the data variables first and then the methods, though this is not strictly required [1].

### 9.2.3 Access Modifiers

Classes make extensive use of two access modifiers: `public` and `private`. These words precede both instance variable declarations and method declarations. Use them to restrict access to certain parts of your class design. For example, we would want the method called `shuffle()` to be `public` so that our class's user can call this method. But we might want the data structure that contains the list of cards to be `private` so that the class's user cannot subvert our interface and change the cards around arbitrarily.

Referring back to the automobile analogy, we want our driver to use the public interfaces such as the steering wheel and brake pedal. We do not necessarily want them to pop the hood and start mucking with the fuel injectors. We'll want these to be private so that only trained professionals who know how the whole car works

[1] Scoping rules will prohibit the use of instance variables in methods that precede the declaration of these variables. Thus it is usually a good idea to list the variables before all the methods.

can make any necessary adjustments. These access modifiers are important for properly implementing the concepts of encapsulation and data hiding.

Most typically, instance variables and other data items are private while methods are public. But there may be some data items, especially constants, that we'll want to make public and there are often a few "housekeeping" methods that we'll keep private since only other methods within the class should call them. Note that by default, all things are `public`. But it is a good idea to use this word explicitly so that your intentions are made clear.

### 9.2.4 Design Decisions

There are many ways to implement a deck of cards in java. There are choices to make regarding the data structures; these will influence how easy or difficult it is to implement certain methods. Now is the time to make some of these decisions.

We decide to use integers for representing individual cards. We know there are 52 cards in the deck so we'll use 0, 1, ... 51 to represent the different cards. Of course our deck class's user won't ever have to know this; this detail is one of the things we'll keep private from the user.

Integers make it easy to compare cards by value. The typical order to cards is to first rank them by kind and then suit. So all 2's are the lowest, followed by 3's, then 4's and so on up to Kings and then Aces which are the highest. Within each kind, the suits are ranked according to clubs, diamonds, hearts, and spades (low to high). This means that the 2 of clubs is the very lowest card (the 0 card) while the Ace of spades is the highest card (card number 51).

Some simple modular arithmetic can be used to convert a card number into its suit and kind. Spend a few moments thinking about how you might do this. Hint: use the mod and div operators along with 4 and 13 (four suits, thirteen kinds). Read on once you have solved this problem yourself.

We can obtain the card's kind by taking its number and dividing by four:

$$kind = cardNum/4;$$

Similarly, the card's suit is obtained by taking its number and "mod"ing by four:

$$suit = cardNum\%4;$$

Of course, this gives us a "suit number" from 0 to 3. We'll have to convert this to an appropriate string using a `switch` statement.

We choose to use an array of `int`'s to store the card numbers. The order of numbers in the array determines the order of the cards in the deck. By rearranging (permuting) the numbers in the array we can shuffle the deck. To deal cards, we can extract them from the front of the array. We'll need an index counter so we can keep track of which card is currently on the top of the deck (i.e. which array location represents the current top index).

Our class is starting to take shape:

```
1  //=========================================================
2  // Matt Kretchmar
3  // August 1, 2005
4  // Deck.java
5  //
6  // Abstract Data Type description should go here.
7  //=========================================================
8
9  public class Deck
10 {
11    // We use integers 0..51 to represent the cards.
12    // kind = cardNum / 4;
13    // suit = cardNum % 13;
14    // Array spots [index .. 51] are used to stored cards.
15    // This implies that the index keeps track of the
16    // current top of the deck.  Array locations
17    // [0..index-1] are unused (cards have already been
18    // removed from the deck).
19    private int cards[];     // cards stored by number
20    private int index;       // index = current top deck
21    public static final int NUM_CARDS = 52;
22
23    //-------------------------------------------------
24    // shuffle
25    // Shuffles the cards in a random order.
26    //-------------------------------------------------
27    public void shuffle ( )
28    {
29    }
30
31    //-------------------------------------------------
32    // draw
```

```
33      // Returns (and removes) the next card in the deck.
34      //-------------------------------------------------------
35      public String draw ( )
36      {
37        return null;
38      }
39
40    } // end of Deck class
```

Notice we have included extensive comments detailing the specifics of our implementation. We should probably include a copy of the ADT document in the comment heading at the top of the class. In addition to the private instance variables, we have also added blank methods for shuffling and drawing; we'll add the code to them shortly [2]. In future installments, we'll remove some of the comments so that the code fits within this book more easily.

We have placed a `public final int` in the instance variable section named `NUM_CARDS`. This is a constant specific to the class (recall the keyword `final` implies a constant). We use the convention for constants of all uppercase letters using an underscore to separate multiple words; this makes it easy for someone reading our code to know that `NUM_CARDS` is a constant variable without having to look for its declaration. We make it public so that users of our class can access this constant.

## 9.3 Constructors

If you were attentive, you might have noticed a problem with our class. In the instance variable section, we *declared* an array to hold the deck. But we did not *allocate* any memory for the array yet; we have a reference that does not yet point to an actual array. In the instance variable section, we can only declare variables. We cannot execute regular java statements including array allocations. So how is it that we can create an array before the user starts calling shuffle and other routines that won't yet work without an array?

The answer is a special method called a constructor. A *constructor* is a method that is secretly called when a new object of this class type is created. The class designer uses the constructor to properly initialize the class right before it gets used. This includes allocating memory for arrays and other object types as well as

[2] The `return null;` statement inside the `draw()` method is to allow our temporary class file to compile. The `draw()` method must return a string, so this statement is necessary to avoid an error. We'll remove it later.

initializing variables with certain values. In our `CardDeck` class constructor, we need to (1) allocate the array, (2) place the cards in the array, and (3) initialize the index to the top of the deck.

A constructor method has the same exact name as the class name. It is a unique method in this respect. Here is our deck class constructor method (we show only the method here though a constructor is typically the first method in the list after the instance variable section).

```
1  //-----------------------------------
2  // Default constructor
3  // Creates a new deck in sorted order
4  //-----------------------------------
5  CardDeck ()
6  {
7    cards = new int[NUM_CARDS];
8    for ( int i = 0; i < NUM_CARDS; i++ )
9    {
10     cards[i] = i;
11   }
12   index = 0;
13 }
```

This is called the *default constructor* since it takes no arguments. You can have other constructors with input arguments if it makes sense for your particular class. We could have a constructor with an input `int` that tells us how many cards to start our deck with (though this seems not in keeping with our common and minimalist goals for the class). Notice the constructor never has a return type.

### 9.3.1 Finishing shuffle() and draw()

Now that we have our constructor finished, we can fill in the code for the `shuffle()` and `draw()` methods.

```
1     //-------------------------------------------------------------
2     // shuffle
3     // Shuffles the cards in a random order by using an exchange
4     // shuffle algorithm.
5     //-------------------------------------------------------------
6     public void shuffle ( )
7     {
8        for ( int i = NUM_CARDS-1; i > index; i-- )
```

```
9         {
10           int spot = (int)( Math.random() * (i-index+1) ) + index;
11           int temp = cards[i];
12           cards[i] = cards[spot];
13           cards[spot] = temp;
14        }
15     }
16     //-----------------------------------------------------------
17     // draw
18     // Return the next card at spot index.  Move index to the next
19     // card.  It is an error to draw from an empty deck.
20     //-----------------------------------------------------------
21     public String draw ( )
22     {
23       if ( isEmpty() )
24       {
25         System.out.println("Error: deck empty");
26         System.exit(1);
27       }
28       int card = cards[index];
29       index++;
30       return cardToString(card);
31     }
```

The `draw()` method is the easiest to follow. Recall that the `index` instance variable is used to hold the position in the array of the current top card in the deck. To draw a card, we need only to access the card at the location specified by `index` and then increase the index to the next array spot. Of course, we'll need to handle the error condition that occurs when a draw is made on an empty deck. Figure 9.3 depicts how the `draw()` method works.

Notice that in `draw()` we call two other methods. Both `isEmpty()` and `cardToString()` are methods that we'll need to define in our class. The `isEmpty()` method will be `public` so that the user can access it too while `cardToString()` will be `private` since this is one of those details that we want to hide from the user. Can you fill in the code for these two methods on your own [3]?

The `shuffle()` method works by making random exchanges between cards in the deck. Do you think each card has the same probability of being permuted to the same destination by this algorithm? This would be an important property for a

[3]The code for these methods is shown at the end of the chapter.

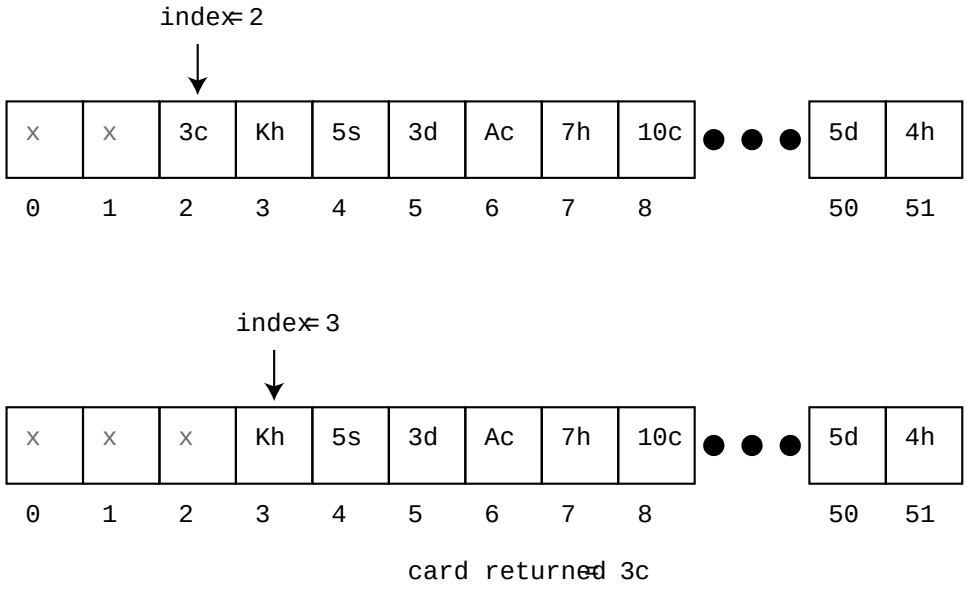



Figure 9.3: `draw()` method

blackjack application otherwise good players might learn to exploit the statistical distribution of cards in our not-so-good shuffling algorithm.

## 9.4 Objects and Memory

Though we have discussed the memory differences between primitive types and objects in the arrays chapter, it is worth revisiting that discussion here since the concept is so critical to the proper design and use of objects.

Suppose we have the following code in our program:

```
int num;
num = 1;
CardDeck deck1;
deck1 = new CardDeck();
```

The variable `num` is of type `int` which is a primitive type. The first statement declares and allocates space for the integer. There is no value in this memory location yet [4]. The second statement assigns the value of 1 to the variable.

[4]Technically speaking, there might already by some latent data in this memory location so `num`

The variable `deck1` is of type `CardDeck` (our newly designed data type). The third statement only declares a reference to some `CardDeck` but does not yet actually allocate any memory for the `CardDeck`. In Figure 9.4, the declaration statement creates the reference box (upper right) but does not yet allocate an object (lower right box). The fourth statement above does the actual allocation. It creates a memory location for the new `CardDeck` object (lower right box in Figure 9.4) and then assigns the reference to "point" to this memory location.

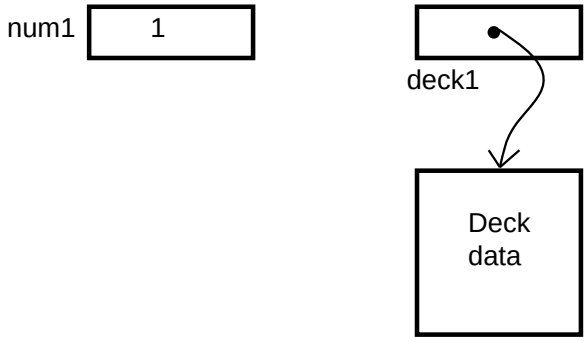



Figure 9.4: memory for objects

Java does automatic memory management. When you allocate space for a variable, java will search computer memory and find some space for you. If this space is for an object, it will set the object variable's reference to point to this space. When you are done using the variable, java will automatically detect that the memory is no longer used and free up that memory space for other variables to use.

### 9.4.1 Copying objects

Consider the following code segment:

```
int num1 = 1;
int num2 = num1;
Deck deck1 = new Deck();
Deck deck2 = deck1;
```

Can you draw a memory model that illustrates these statements? Figure 9.5 shows the resulting memory model. Notice that two separate variables for the

probably already has a value. In fact, java initializes new integers to 0 but you should not rely on this. Instead explicitly initialize the variable to 0 with an assignment statement.

integers are created. The assignment statement in line 2 above *copies the value* from one variable to the other. You can change the value of one variable (say `num1`) and it won't affect the value of the other variable (`num2`).

However, there is only one `CardDeck` object. The assignment statement in line 4 above merely *copies the reference* of one variable to the other `CardDeck` variable. Now both variables refer to the same piece of memory. If you make changes to `deck1`, the same changes will be made to `deck2` as well because both variables share the same object. This is a critical difference between primitive types and object types.

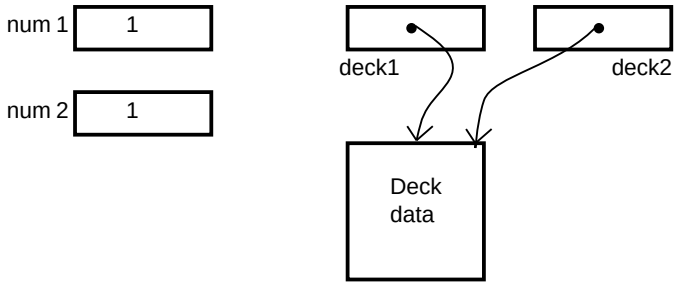



Figure 9.5: copying an object ?

If you do indeed want to make a separate copy of the object for a second variable, then you must design some type of `copy()` method in the `CardDeck` class. This method would create a brand new `CardDeck` object and copy all the appropriate values into this newly created object.

### 9.4.2 Comparing objects

Notice that if you do indeed have two separate `CardDeck` objects (through two separate allocation statements), you cannot compare them using the `==` operator. While `num1 == num2` will successfully compare the values of these two variables, `deck1 == deck2` will not compare the decks. This latter statement only compares the references. If both variables use the same reference (i.e. point to the same object), then this statement will return true. However if both variables point to different but identical `CardDeck` objects, then the `==` operator will return false. This is why you use the `.equals()` operator when comparing two strings. Most useful classes will implement a `.equals()` operator so that the class's user can compare two objects.

## 9.5 Extending the Class: Common Methods

There are certain methods that are implemented in many classes. They have evolved standard names within the java language. In this section, we examine three such methods.

As hinted in our previous section's discussion, we need a `copy()` method if we are to be able to make copies of our `CardDeck` objects. Also important is an `equals()` method for comparing to `CardDeck` objects. Finally, we will implement a `toString()` method so that we can convert our deck to a string for printing purposes.

Note that "equals" and "toString" are standard names for each of these operations. There are times when java will attempt to call these methods (if they exist) automatically. For example, suppose your class user tried to compile/execute `System.out.println(deck);`. Secretly, java attempts to convert `deck` to a string by calling the `CardDeck` class's `toString()` method. Thus it is a good idea to implement at least this method. We also implement `equals()` and `copy()` for illustration purposes though their use might not be practical enough to warrant inclusion in our ADT.

The `copy()` method needs to accomplish two important tasks. First it needs to create an entirely new `CardDeck` object. Second it needs to copy all the instance variable data from the current `CardDeck` object to the newly created one. A reference to the new `CardDeck` is returned. The complete code listing for `copy()` is shown in the next section.

For `equals()`, we need to compare first the `index` of both decks. If these are the same, then we compare card for card in the two arrays. Any sign of a nonconsistency between the two decks and we immediately return `false`. If we find no such inconsistency, then we return `true` indicating the two decks are the same in every respect.

Finally for `toString()` we create an empty string and append cards one at a time from the array to the string.

## 9.6 The complete `CardDeck` class

Here we list the completed `CardDeck` class. We have added some other public routines (namely for starting over with a sorted deck) and added some internal private methods to accomplish some useful tasks. We have left out many of the comments to prevent the class from growing too large to print in this book. But you should comment your class bountifully including adding the whole ADT description to the top of the class.

Notice how we have reduced instances of repeated code by calling methods within the class. For example, the constructor now calls the `initialize()` routine since the constructor needs to accomplish all these same tasks. Why have a duplicate section of the same code within the constructor when you can call a method and save typing/space?

```
1  public class CardDeck
2  {
3     private int cards[];
4     private int index;
5     public static final int NUM_CARDS = 52;
6
7     //-----------------------------------------------------------------
8     // Default constructor
9     // Creates a new deck in sorted order
10    //-----------------------------------------------------------------
11    CardDeck ()
12    {
13       cards = new int[NUM_CARDS];
14       initialize();
15    }
16
17    //-----------------------------------------------------------------
18    // initialize
19    // Places the cards in sorted order first by suit, then by kind
20    // in ascending order (aces high).
21    // 2c, 3c, ... Kc, Ac, 2d, 3d, ..., As
22    //-----------------------------------------------------------------
23    public void initialize ( )
24    {
25       for ( int i = 0; i < NUM_CARDS; i++ )
26       {
27          cards[i] = i;
28       }
29       index = 0;
30    }
31
32    //-----------------------------------------------------------------
33    // shuffle
34    // Shuffles the cards in a random order by using an exchange
```

```
35      // shuffle algorithm.
36      //-----------------------------------------------------------------
37      public void shuffle ( )
38      {
39         for ( int i = NUM_CARDS-1; i > index; i-- )
40         {
41            int spot = (int)( Math.random() * (i-index+1) ) + index;
42            int temp = cards[i];
43            cards[i] = cards[spot];
44            cards[spot] = temp;
45         }
46      }
47
48      //-----------------------------------------------------------------
49      // toString
50      // Converts the deck to a string using 2,3,4...10,J,Q,K,A for kinds
51      // and c,d,h,s for suits.
52      //-----------------------------------------------------------------
53      public String toString ()
54      {
55         String deckString = new String();
56         for ( int i = index; i < NUM_CARDS; i++ )
57         {
58           deckString = deckString + cardToString(cards[i]) + " ";
59         }
60         return deckString;
61      }
62
63      //-----------------------------------------------------------------
64      // cardToString
65      // Converst a card number to a string for that card.
66      //-----------------------------------------------------------------
67      private String cardToString ( int cardNumber )
68      {
69        String cardString = null;
70
71        if ( cardNumber >= 0 )
72        {
73          int suit = cardNumber % 4;
74          int type = cardNumber / 4;
```

```
75          cardString = new String();
76
77          switch(type)
78          {
79            case 9 : cardString = cardString + 'J';
80            break;
81            case 10: cardString = cardString + 'Q';
82            break;
83            case 11: cardString = cardString + 'K';
84            break;
85            case 12: cardString = cardString + 'A';
86            break;
87            default: cardString = cardString + (type+2);
88            break;
89          }
90
91          switch(suit)
92          {
93            case 0 : cardString = cardString + 'c';  // clubs
94            break;
95            case 1 : cardString = cardString + 'd';  // diamonds
96            break;
97            case 2 : cardString = cardString + 'h';  // hearts
98            break;
99            case 3 : cardString = cardString + 's';  // spades
100           break;
101         }
102
103       }
104       return cardString;
105     }
106
107     //-----------------------------------------------------------------
108     // drawCard
109     // Return the next card at spot index.  Move index to the next
110     // card.  It is an error to draw from an empty deck.
111     //-----------------------------------------------------------------
112     public String drawCard ( )
113     {
114       if ( isEmpty() )
```

```
115       {
116         System.out.println("Error: deck empty");
117         System.exit(1);
118       }
119       int card = cards[index];
120       index++;
121       return cardToString(card);
122     }
123
124     //-----------------------------------------------------------------
125     // getNumberCards
126     // Returns the current number of cards left in the deck.
127     //-----------------------------------------------------------------
128     public int getNumberCards ( )
129     {
130       return NUM_CARDS - index;
131     }
132
133     //-----------------------------------------------------------------
134     // isEmpty
135     // Returns true if deck is empty, false otherwise.
136     //-----------------------------------------------------------------
137     public boolean isEmpty ()
138     {
139       return getNumberCards() == 0;
140     }
141
142     //-----------------------------------------------------------------
143     // equals
144     // Returns true if decks match completely, false otherwise.
145     //-----------------------------------------------------------------
146     public boolean equals ( CardDeck other )
147     {
148       if ( other.index != index )
149         return false;
150
151       for ( int i = index; i < NUM_CARDS; i++ )
152       {
153         if ( cards[i] != other.cards[i] )
154           return false;
```

```
155       }
156
157       return true;
158     }
159
160     //-----------------------------------------------------------------
161     // copy
162     // Returns a copy of the Deck object.
163     //-----------------------------------------------------------------
164     public CardDeck copy (  )
165     {
166       CardDeck newDeck = new CardDeck();
167       newDeck.index = index;
168       for ( int i = index; i < NUM_CARDS; i++ )
169         newDeck.cards[i] = cards[i];
170
171       return newDeck;
172     }
173
174   } // end of CardDeck class
```

## 9.7 Summary

This chapter is a brief introduction to classes and objects. The topic is extremely extensive and diverse both technically and philosophically. Indeed, professional computer scientists who have programmed with objects for years are still debating many of the key philosophical issues. Newly trained programmers can only develop a feel for these debates after some extensive practice designing and using their own objects.

From this chapter, you should be familiar with the key terms such as *object* and *class*. You should know the syntax for declaring and creating a class in java (though you will probably need to look back at existing classes for syntax help until you develop several of your own). Most critical is the understanding of how memory is treated differently between objects and primitive types. We also introduced constructors and several other common methods for printing, copying and comparing objects.

As you design your own objects, be sure to start to think about style. There are design decisions to make; poor decisions result in complex, messy, and inflexible objects that are error prone. Good design decisions promote flexibility and expand-

ability. They almost always seem to have fewer errors to fix as well. Most critical to the design process is to spend extensive time thinking about your class before you ever start to write code. Think about programming *elegantly*! You should view your class designs with the same satisfaction that an artist might receive from a good painting.

## 9.8 Exercises

**Exercise 9.1** *List three ways in which user-defined objects differ from primitive types.*

**Exercise 9.2** *Explain why there is both a declaration step and an allocation step when using object variables. What does each step accomplish?*

**Exercise 9.3** *Define the following terms:*

- *class*
- *object*
- *abstract data type*
- *instance variable*
- *constructor*

**Exercise 9.4** *Define the following terms:*

- *encapsulation*
- *data hiding*
- *inheritance*

**Exercise 9.5** *What are the absolute rules for choosing names for java classes? What are the conventions for choosing class names and why are they important?*

**Exercise 9.6** *What does the keyword* `private` *do to an instance variable? To a method?*

**Exercise 9.7** *What does the keyword* `public` *do to an instance variable? To a method?*

**Exercise 9.8** *In this exercise we'll trace through the execution of the* `shuffle()` *method to see how it works. To make our life easier, suppose we have a deck where* `NUM_CARDS = 10` *instead of 52. Suppose you start with the array given below (again cards are stored as integers 0..9). Suppose that* `index = 0` *so that we haven't drawn any cards yet. Run the* `shuffle()` *method and show what happens to the array each pass through the outer for loop. You can make up random numbers for the calls to* `Math.random()` *but be sure to list them along side the array to help illustrate the results.*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|

**Exercise 9.9** *What is the purpose of including a method named* `toString()` *in your class design?*

**Exercise 9.10** *What is the purpose of including a method named* `equals()` *in your class design?*

**Exercise 9.11** *Why can't the comparison operator* `==` *be used to see if two objects are the same?*

**Exercise 9.12** *Consider the following block of code:*

```
CardDeck deck1 = new CardDeck();
CardDeck deck2 = new CardDeck();
boolean same = deck1.equals(deck2);
```

*What will be in the boolean variable* `same` *after execution? Will it always be the same result everytime you run the code?*

**Exercise 9.13** *In the game of Go Fish, each player is dealt seven initial cards from a shuffled deck. Programmer Bob attempts to implement one of these deals with the following block of code. Does this code correctly deal seven cards for one of the Go Fish players? Explain.*

```
for ( int i = 0; i < 7; i++ )
{
CardDeck deck = new CardDeck();
deck.shuffle();
System.out.println(deck.draw());
}
```

**Exercise 9.14** *In Vegas, casinos use a seven deck stack for playing Blackjack (to make it harder for the patrons to count cards). Make a new class called* `VegasDeck` *that works exactly the same as our* `CardDeck` *class except it contains seven full decks (that's* $7 \times 52 = 364$ *cards with seven copies of each card).*

**Exercise 9.15** *Design an abstract data type (no code) for a class that a bank can use to handle a generic car loan.*

**Exercise 9.16** *Building from your ADT in the previous exercise, design a class for the car loan. Some of the design decisions are made for you here:*

- *List the loan by social security number only. To make life easier, you don't need to include anything else about the client such as name, address, phone, etc.*
- *Make the interest rate a constant in your program of 8%.*
- *Allow the term to be selected from among 24, 36, 48, and 60 month periods.*
- *Keep track of the current balance.*
- *Have a default constructor that assumes each person is borrowing $20,000 (initial principle).*
- *Have a second constructor that specifies the starting principle.*
- *Have a method that allows the banker to enter/change the social security number.*
- *Have a method that can be called at the end of each month to update the balance according to the interest rate (remember to divide the annual interest rate by 12 for use as a monthly interest rate).*
- *Have a method that allows the balance to be lowered when the user supplies a payment.*
- *Have a method that returns the current balance.*

**Exercise 9.17** *People shuffle decks quite differently than our exchange algorithm. People split the deck into two pieces (a top half and a bottom half). Then they interleave some cards from each half. That is, they take a few cards from the top half, then a few from the bottom, then a few more from the top, then a few more from the bottom, and so on until the cards are gone. The relative orders of cards of*

*cards in each half does not change on any one shuffle operation. Watch someone shuffle or shuffle yourself and think hard about how the operation works.*

*Implement this type of shuffle algorithm in our* `CardDeck` *class with the following steps:*

1. *Split the deck into two equal halves: a top half and a bottom half.*
2. *Pick a random number from 0 to 4, select this many cards (in order) from the top of the top half and place them on the new pile (on the bottom of the new pile).*
3. *Pick a random number from 0 to 4, select this many cards (in order) from the top of the bottom half and place them on the bottom (underneath) of the new pile.*
4. *Continue alternating steps 2 and 3 until all the cards are placed in the new pile.*

*Note, you will need to call this routine at least 7 times in a row to make sure the deck is fairly well shuffled.*

# Chapter 10

# Files

We have seen a variety of ways to input data to Java programs. One way to do this is to use the `CS171In` methods which allow one to enter several different kinds of data, such as integers, doubles, and strings by typing them at the keyboard. Another way is to use dialog boxes, again using the keyboard to enter the data. These are useful ways to enter data, particularly when the amount to be entered is small. However, if we have a program that requires a large data set or if we have a program that might be used for a lot of different data sets, it would be useful to have a way of telling Java that the input will be found in a file.

## 10.1 Reading from a File

Suppose for example that each student in a class has a text file of their quiz scores and would like to get the average grade for those quiz scores. Since the scores are already stored, it would eliminate a time consuming step to have the computer get the scores directly from the file, rather than have the student look them up and then enter them at the keyboard.

Here we see the contents of a file named "scores.dat" with 10 quiz scores in it. As in our previous examples, the lines are numbered to make it easy to refer to them.

---

```
1   78
2   82
3   94
4   69
5   88
6   79
```

```
 7  90
 8  85
 9  87
10  84
```

---

The scores are written in the file one score per line, which will allow us to read them one line at a time.

Java uses two class types for processing data from a file — one type called `FileReader` and a second called `BufferedReader`. A `FileReader` object can establish an association with a text file. This is called "opening" the file. The BufferedReader is constructed using the `FileReader` as a building block. Each line of the file is interpreted as type `String`, and so, just as we do with Dialog Box input, if we want to use the entries as numbers, the program will need to convert them to integers or real numbers as their use requires.

In the next example, we see a program that reads scores from the file called "scores.dat" and finds the average value of the scores therein. The ".dat" extension is used to suggest "data." It is also permissable to use a ".txt" extension after the file name to suggest text.

---

```
 1  // The "FileAvg" class.
 2  // Input a sequence of scores entered one to a line
 3  // from a file whose name is to be read in from the keyboard
 4  // and find their average.
 5
 6  import java.io.*;
 7  import java.util.Scanner;
 8
 9  public class FileAvg
10  {
11      public static void main (String [] args) throws IOException
12      {
13
14
15   String fileName, line = " ";//a variable to hold the name of the file to be proces
16        //a variable to read in one line from the file
17   int score = 0;// to hold the converted string from the file
18   int sum = 0, count = 0;//to hold the cumulative sum and to count how many numbers
19   Scanner consoleIn = new Scanner(System.in);
20
21   System.out.println ("What is the name of the file of integers? ");
22   fileName = consoleIn.nextLine();//read the name of the file
```

```
23
24    FileReader inputFile = new FileReader(fileName); //open the file requested
25    BufferedReader inputReader = new BufferedReader(inputFile);//initialize the new FileReader
26
27    line = inputReader.readLine (); //Read one line of characters from the file.
28
29    while (line != null)                    //File is terminated by a null, i.e. at the end of t
30    {
31        score = Integer.parseInt (line);  //Change one line of characters to an integer.
32        count=count+1;//increment the counter
33        sum = sum + score;//update the cumulative sum
34        line = inputReader.readLine ();       //Read next line.
35
36    }
37    //end the loop
38    inputReader.close(); //close the file
39   System.out.println ("The average of your " + count + " scores is " + (double)sum/count);
40       } // main method
41   } // FileAvg class
```

---

The first line of the program imports the types and methods needed by Java to carry out file manipulations, while the second line allows us to use the `Scanner` class. Something new appears on the line where the `main` method begins, `throws IOException`. This is what we write in Java to indicate that there are some exceptional conditions that might arise when input or output are attempted. For example, it might happen that the user wants to read from a file, but makes a typographical error when entering the name of the file. The user needs a message alerting him/her to the fact that the file requested does not exist. When the user sees the error message, he/she can retype the name of the file, correcting the typo.

Inside the main method we declare a `String` variable called `fileName` to receive the name of the data file to be entered by the user at the keyboard. The `int score` will be used to hold each score as it is read from the file. The variable `sum` will act as an accumulator adding each score to the existing sum as the score is read. `count` keeps track of how many scores have been read so far. When all the scores have been read, the average can be computer using the value of `count`.

We use our usual keyboard input method to get the name of the file to be used. It will be necessary for the user to type the full directory path name for the file. For example, if the user has a file named scores.dat in a directory on the C drive named History, when prompted for the file name, the user should input `C:/History/scores.dat`.

The declaration of `input` to be a FileReader both identifies `inputFile` to be the given class type and opens the file that the user has specified. The Buffere-

dReader declaration of `inputReader` also both identifies the class type and associates the new instance with the `inputFile`. When these two declarations have been executed, the program is ready to read from the designated file. The data from the file is read one line at a time using the `readLine` method, which is provided by the BufferedReader class.

Once a line has been read, the `while` loop begins to process the file. The check for stopping the loop is the boolean expression `line != null`, which is equivalent to checking to see whether the line is empty. In other words, if the line read in has anything but the null string in it, the loop continues, but once the end of the file is reached, the loop will stop. Hence, we do not need to know how many entries are in the data file in order to process it, but we do need to count the number of entries so we can compute the average when the sum has been found.

The steps of the loop that are to be repeated include converting the string that was read into an integer. The `Integer.parseInt(line)` accomplishes this at line 31. The count is incremented to indicate that another value has been read. The sum adds on the newly read value of `score`, possible because the input string has been converted to an integer. The last step of the loop is the reading in of another line from the file.

When the boolean expression that controls the loop becomes false, i.e. the file has been completely read, the loop stops and the file is closed. Finally, the average is computed and printed.

## 10.2 Writing to a File

One way to prepare a data file that can be processed from a Java program is to use any editor and put data elements one per line in the editing window. Then save the file with a .dat or .txt extension. However, it is also possible to prepare a file by using a Java program. One advantage of writing the file from a Java program is that helpful messages can be printed for the user allowing the user to make entries directly into the Java program which can store them in a file named by the user.

To accomplish this task, we need writing counterparts to the `FileReader` and `BufferedReader` classes that we used for reading from files. From the same `java.io.*` classes we can get just what we need. To create and open a file for writing there is a class called `FileWriter` which is the writing analog to FileReader. Instances of class type `FileWriter` can establish a connection to a file for writing. The writing counterpart to `BufferedReader` is `PrintWriter` which allows a user to write to a file using the methods named `print` and `println` and which act the same as the `print` and `println` we have used previously in `System.out` output.

In our next example we allow the user to designate a file name to receive the data that is input from the keyboard. When the program ends, the data will still exist in the file. As we know, data that is entered from the keyboard, but not written to a file, disappears when the program ends, because that data is kept only temporarily in main memory. Data written to a file is kept on the hard drive which retains its data even when the computer is turned off.

---

```
1  //This program prompts a user for a file name to store some scores.
2
3  import java.io.*;
4  import java.util.Scanner;
5
6  class outFile
7  {
8    public static void main(String[ ] args) throws IOException
9    {
10     String fileName; //the name of the file
11     int numScores = 0; //the number of scores
12     String score; //a variable to hold each score as it is entered
13     Scanner consoleIn = new Scanner(System.in);
14
15     System.out.println("How many scores do you have?   ");
16     numScores = consoleIn.nextInt(); //get the number of scores
17     consoleIn.nextLine();
18     System.out.println("What do you want to call your file of scores?  ");
19     fileName = consoleIn.nextLine(); //get the name of the file
20
21     //open the file
22     FileWriter fwriter = new FileWriter(fileName);
23     PrintWriter outFile = new PrintWriter(fwriter);
24
25     //get the data and write it to the file
26     for (int i = 1; i <= numScores; i++)
27     {
28       System.out.println("Enter a score:   ");
29       score = consoleIn.nextLine();
30
31       outFile.println(score); //read the score as a string
32       }
33
34     outFile.close();
35     System.out.println("Data written to file.");
36     }
37     }
38
39
```

As before we begin by importing `java.io.*`. In the main method we declare a `String` to hold the name of the file we want to produce. The delcaration `String score` sets up a variable to hold each score as it is entered. The program prompts the user for the number of scores to be placed in the file and then asks what the user wants to call the file.

Once the name of the file has been read, a `FileWriter` varible is declared and initialized. A `PrintWriter` is declared using the `FileWriter` variable as parameter and the effect of these two declaratons is to open a file of the required name for writing. The `for` loop reads the scores, as `Strings`, one at a time from the keyboard and then writes them as strings, one per line, to the file. When the loop has read and stored all of the scores, the file is closed. A message informs the user that the data has been written to the file.

## 10.3 exercises

**Exercise 10.1** *Write a program to allow a user to enter as many car prices as the user wants and then stores those prices to a file called "carPrices."*

**Exercise 10.2** *Write a program that reads from the "carPrices" file and finds the average price of a car, the maximum cost, and the minimum cost.*

# Chapter 11

# SearchandSort

Two of the most common and most important actions that computers do for us are searching stored materials to find what we need and putting our stored materials into whatever order we want them. Of course, to be really helpful, these activities need to be fast. For example, when we want to find books covering a particular subject from our media center, we want to be able to tell the searching program what the topic is and then to get a list of available materials quickly. Often, seaching facilities will allow the user to tell what order the user wants results displayed. In the media center example, the usual default is to display the findings in reverse chronological order, but users can choose to have them displayed in other ways. For example, when doing a web search, we usually want the items to be displayed in their order of popularity, meaning that those items which have been used by others most often will be shown first.

## 11.1 Searching

### 11.1.1 Searching Randomly Stored Data

Suppose we have a file of healthful foods and we are just about to choose an apple as a snack. We want to confirm that apple is on the list, so we need to search the file to see if apple is there. Assuming that the food names are stored as they were tested by the FDA for healthfulness, they do not appear in any particular order. So our search will need to start at the beginning and move through the food names to see if we can find apple. We call such a search ”linear” because, if we conceptualize the data as being laid out in a line, the linear search moves through the data by going along the line.

The following example shows a linear search that searches an array of integers

that have been entered randomly. The method returns the index of the first position where the desired value (given on the paramter list) is found. If the value is not found, then the method returns -1. The calling program can then generate an appropriate message.

---

---

When the data is stored in a random fashion, we have no choice but to examine every entry of the designated items in the order in which they are kept. To shorten the search, we could stop upon finding the value we are searching for. You will do this as an exercise. Another interesting problem is to find all the positions that hold the given data. Still another is to count how many such positions there are. For example, if you wanted to check a random number generator to see how good it is, you could generate a large file of random numbers and then count how many of each there are. We would expect a fairly uniform distribution of the numbers. Unusually large or small numbers of a particular value might give us reason to believe that the random generator in not doing a proper job.

### 11.1.2 Searching Ordered Data

Let's now consider how we might speed up a search in the case where the data is stored in a particular order. A good example of this idea is a phone book. In a phone book the names are written in alphabetical order, so if we are looking for some name, such as Miller, we would probably not start at the beginning of the book, but rather, open to approximately the middle. At the middle we would check to see which names are there and if the one we want isn't on that middle page, we would decide whether to go back toward the front of the book or toward the end of the book, depending on whether the name we want comes before or after the ones on the middle page.

We can apply this idea to computer data that is stored in numeric or alphabetic order. Give a sequence of data stored in order, if we need to look for a particular value, we can look first at the middle element and then, if the middle element is the one we want, stop. If the one we want is not the middle one, we can then apply the same approach to half the entries, since we can tell whether we need to look in the first half of the data or the second half.

Here we will see two versions of binary search methods. The first is an iterative version. In this version, the method `itbinsearch` receives four paramaters.

The first is `int[] v`, which is an array of integers to be searched. The second parameter is `int w`, the value to search for. The third, `int first` is the index of where to begin the search, and the fourth, `int last` is where to stop the search. When the method is called, `first` will probably be `0`, the first index into the array, and `last` will be the one less than the size of the array.

The major part of the method is a `while` loop that is controlled by whether the value of `first` is less than `last`. Of course, unless the array has only one entry, that will be the case when the method is first called, and so the loop body will execute. The value of the middle index of the array is computed, `mid`. If the entry in the array at that index is `w`, then the method returns `mid`, the index of `w`. Otherwise, the values of `first` and `last` are updated so that the search will continue only in half of the remaining unexamined entries. The same process is iterated until all appropriate parts of the array have been examined. If the required vale is found, the index of it is returned. Otherwise, a value of -1 is returned indicating that the value was not found anywhere in the array.

```
1  int binarySearchIterative (int[] v, int w, int first, int last)
2      {
3        int mid=0;//variable to hold the value of the middle index
4        while (first <= last)//keep repeating the loop as long as long as first is less than la
5        {
6          mid = (first+last)/2;//find the midpoint
7          if (w==v[mid])//if the needed value is at the middle index, return
8            return mid;
9          else if (w < v[mid])//if the value you want is less than the middle value,
10           //search only the first half
11           last = mid-1;
12         else
13           first = mid+1;//if the value you want is greater than the middle value,
14         //search only the upper half.
15       }
16       return -1;//if the value is never found, return -1
17      }
18
```

### 11.1.3 Recursive Methods

There is another common way to write a binary search method. Many programmers think that this new way, called recursive, is a natural way to write the binary search. A recursive method is one that calls itself. When doing a binary search, once we

have checked the middle element and figured out whether to look next only in the lower half or only in the upper half, we want to carry out exactly the same steps we did with the whole sequence, but apply those steps only to half of the elements. The only changes we want to make are the values of `first` or `last`.

So, informally, what we want to do is to call the method with `0` as `first` and one fewer than the number of elements as `last`, find the middle value, and then change either `first` (in the case where we need to look only in the upper half) or `last` (when we need to search the lower half). The rest of the work is to apply the same algorithm to whichever half we have identified. We do that by making the call to the method with the updated values for `first` and `last` on the parameter list.

The big question that remains is "How can we get this process to stop?" This is always an important challenge in every recursive procedure. The way we address it is to put a check right at the beginning of the method that checks to see if the value of `first`is greater than the value of `last`. When that happens, the method just stops and returns whatever value it currently has in `index`. Notice that we used the clause "When that happens," rather than "If that happens." That's because eventually, since at each call to the method, either the value of `first` increases or the value of `last` decreases. Hence, at some point `first` will overtake `last`.

Here is the code for doing a binary search in the recursive style:

---

```
1  int binarySearchRecursive (int[] v, int w, int first, int last)
2       {
3         int mid=-1;// variable to hold the middle index
4         int index=-1;//variable to hold the index of the value
5
6        if (first > last)// if the value of first is larger than last, just return th
7          return index;
8        else
9        {
10         mid = (first + last)/2;//find the new mid point
11
12         if (v[mid] == w)//if the value needed is at the midpoint, return that index
13         {
14           index = mid;
15
16           return index;
17       }
18
19         else if (w < v[mid])//if the value is less than the one at the index, repea
20           //after first adjusting last
21         {
```

```
22            index = binarySearchRecursive(v, w, first, mid-1);//the recursive call
23          }
24          else if (w > v[mid])//if the value is less than the one at the index, repeat the proc
25            //after adjusting the first.
26          {
27            index = binarySearchRecursive(v, w, mid+1,last);//the recursive call
28
29          }
30        }
31
32        return index; //return the index
33
34      }
```

---

## 11.2 Sorting

In order to be able to use either the iterative or the recursive binary search, both of which shorten the search significantly, it is necessary that the data be stored in some designated order, such as small to large, large to small, alphabetical, or some other user specified order. This leaves us with the problem of how the data can be arranged in order. Of course, one way to get the data in order is to enter it in ordered form, but this means that someone has to get it in order before data entry takes place. Since putting things in order is a task for which it's easy to write an algorithm, it would be a foolish waste of time for a human to do the ordering.

As computing has developed over the years, several algorithms for sorting data have been found, each with different performance characterisitics. Some algorithms are designed to work best on random data, while others save time when the data is partially ordered or in some particular form. Here, we will examine two ways for sorting data, both intended for use when the data have been entered with no special pattern.

### 11.2.1 Bubble Sort

The first algorithm we will consider is one called "bubble sort." In this case we assume we have a sequence of data elements. The strategy is to make several passes through the sequence of data, so that each pass results in the next highest entry "bubbling" to the appropriate position. This means that after one pass through the data, the largest element will be at the end of the sequence. After the second pass, the second largest entry will be next to the end, etc. Of course, one can change

| 1 | 2 | 3 | 4 | 4 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 4 | 7 | 2 | 3 | 5 | 4 | 1 | 3 |
| 4 | 7 | 2 | 3 | 5 | 4 | 1 | 3 |
| 4 | 2 | 7 | 3 | 5 | 4 | 1 | 3 |
| 4 | 2 | 3 | 7 | 5 | 4 | 1 | 3 |
| 4 | 2 | 3 | 5 | 7 | 4 | 1 | 3 |
| 4 | 2 | 3 | 5 | 4 | 7 | 1 | 3 |
| 4 | 2 | 3 | 5 | 4 | 1 | 7 | 3 |
| 4 | 2 | 3 | 5 | 4 | 1 | 3 | 7 |

Figure 11.1: First Pass in a Bubble Sort

"largest" to "smallest" for reverse order or rephrase in terms of alphabetizing if the data consists of text material, rather than numbers.

Having seen the overall strategy, we now look at the detailed description of a single pass through the data. During a pass, we begin at the first position in the sequence and compare it to the second element. If those two elements are not in order already, then we excange them, getting those two elements in the right order. Next, we compare the element now in the second position to the one in the third position and, again, exchange them if they are not already in the right order. We continue comparing pairs of entries until we have reached the end of the sequence. Since each comparison in this pass causes the larger of the two elements to move into the correct position for the two entries being compared, by the time the pass is completed, the largest of the elements being compared is in the correct position relative to the entire sequence.

To see how this works, let's look at a sequence of eight integers to see what happens during the first pass thorugh them as shown in Figure 11.1. The top line of the table has the indices of the entries. The next line shows the original data.

Note that the first comparison results in no change to the order, since 4 is already less than 7. However, when the next comparison is made between 7 and 2, those two values need to be exchanged to get them into the correct order. Subsequent comparisons and exchanges in this first pass thorugh the data result in 7 landing at the end of the sequence, just where it belongs. This figure illustrates two important results: The first pass makes sure that the largest element is in the righr place, and one pass is not enough to get all the elements into the right positions.

Be sure to carry out a second pass on the given data to confirm that after the second pass, the 5 ends up where it belongs. Continue making passes until all the elements are in place.

Now we need to plan how to implement our algorithm. A natural way to store a sequence is in an array, so we will set up an array to hold our entries. Rather than limiting ourselves to eight entries, as we have seen previously, we can allow the number of entries to be filled in by the user, hence making our program flexible enough to handle any size sequence.

Using a loop we can write code to compare array entries to each other in pairs, exchanging when necessary. But we have already seen that a single loop through the array will succeed in getting the largest element in place, but does not guarantee that the rest of the entries are in order, so we need to repeat this loop mulitple times, i.e., we need a nested loop.

To promote efficiency, we observe that, once the last element in in place, there is no need to look at it ever again. Similarly, once the second form the largest is in its place, there is no need to look at it again. This means that each pass through the data can be shorter than the previous pass.

Here is some code to implement this idea:

```
1   void bubbleSort(double [] v, int n)
2       //bubbleSort method receives an array of doubles and an integer telling howmany elements
3       //and sorts those doubles into numberic order from smallest to largest.
4       {
5         int i, j = 0;// counters
6         double temp = 0;//to hold a value temporarily
7         for (i = n-1; i >= 0; i--)//start at the end of the array and work downward
8         {
9           System.out.println("The value of i is  " + i);
10          for (j = 0; j < i; j++)//start at the beginning of the array and work upward
11          {
12            System.out.println("the value of j is  " + j);
13            if (v[j] > v[j+1])//check each pair to see if any exchange should take place
14            {
15              //swap values if necessary
16              swapEntries(v, j, j+1);
17            }
18          }
19        }
20
21      }
22
23  static void swapEntries(double [] A, int i, int j)
24    {
25      double temp = A[i];//set up a temporary storage place
26      A[i] = A[j];//put the value at index j into the place at index i
27      A[j] = temp;//put the temporarily saved value into the place at index j
28    }
```

In the example, there are two methods. One is the method that does the bubble sort, and the other is one which the bubble sort calls to do the swapping of values,in this case, doubles, that need to be exchanged. The `SwapEntries` method takes an array and two indices and exchanges the values at the given indices.

The `bubbleSort` method uses the outer loop to start at the last entry, and after the inner loop puts the largest element into that last position, moves downward so that the inner loop puts the next largest entry into the second from the last position, etc. The outer loop works its way from the end of the array down to the beginning, each iteration putting one more element into the correct position. When the outer loop stops, all the entries are in the correct position.

The bubble sort is not known for speed and efficiency. For example, in the extreme case where all the elements are already in the required order, the method in our example continues to go through the array, checking to see if any values need to be swapped. Multiple passes are made, each resulting in no change, thereby wasting time. So one way to improve the bubbble sort is to stop the process of checking for elements out of order if a pass through the data results in no exchanges, indicating that all of the elements are now in order. Of course, in the worst case, when the elements are in reverse order, all of the passes must be made to get the correct ordering. If there are *n* elements, there may need to be as many as $n - 1$ passes for each, resulting in approximately $n^2$ comparison operations to get the sorting completed.

### 11.2.2 Selection Sort

In the bubble sort we compared pairs of data elements and exchanged those that were out of order, hence bubbling the largest value to the end of the sequence. Ignoring the element(s) already in the right place, we continued that process, bubbling up the largest among the remaining elements until all the data were in order. We turn now to another strategy for sorting. Given a sequence of data to be put in order, we start at the beginning of the sequence and hunt for the smallest element among the data. We then exchange it with the first entry in the sequence. At this point we know that the first entry is in the right place, so we can ignore it and look at the remaining data. Starting with the second entry we examine the sequence and find the smallest element among those entries and exchange it with the second element. We continue this process until we have placed the second from the last entry. The last entry, by default, is in the right position.

Here is an implementation of this strategy, called the Selection Sort, written in Java:

```
1   void swapEntries(int[] v, int i, int j)
2   // exchange the values at the ith and jth entries of v
3   {
4       int temp = v[i]; //save the ith value
5       v[i] = v[j]; // replace the ith value with the jth value
6       v[j] = temp; //put the value saved in temp (the original ith value) in the jth position
7     }
8
9
10  int getMinIndex (int [] v, int first, int last)
11      //find the index of the smallest element starting at first and
12      //ending at last
13      {
14   int minInd = first;//start the min index at first
15   int i = first;//start i at first
16   while (i <= last)//repeat the loop from i to last
17   {
18     if (v [i] < v [minInd])//check to see if any array values
19      //are less than the value at the minInd
20    minInd = i;//if so, replace minInd with the newly found minimum
21         //index
22       i = i + 1;//increment i
23   }
24   return minInd;//return the index of the smallest element between
25     //first and last
26      }
27
28  void selectSort (int [] v, int n)
29      //put the n values of v in order using the selection approach
30   {
31    int i = 0;//to use as a counter
32    int minI = 0;//to hold the index of the smallest element
33
34    for (i = 0 ; i < n ; i++)//repeat for each element in ithe array
35    {
36       minI = getMinIndex (v, i, n);//use the method getMinIndex to find out where
37       //the smallest element is, starting at the ith entry
38       swapEntries(v, i, minI);
39       //put the smallest entry into the right position
40
41    }
42
43  }
44
45
```

The `swapEntries` method is almost the same as the method by the same name in the bubble sort example. The only difference here is that the entries are integers, rather than doubles. We note that we have broken the selection sort strategy into two methods. The first method, `getMinIndex` finds the index of the smallest element in the array `v`, starting with the element in the position `first` and looking at all the entries in `v` up to position `last`, the end of the array. Of course, when the `selectSort` method calls `getMinIndex` for the first time, `first` will be given the value `0`, since no members of `v` have been checked for order yet.

The `selectSort` method initializes the variable `minI` to be 0, because we start by assuming that the smallest element of `v` needs to go at the 0th position. The `for` loop checks through `v` and finds the index of the smallest integer in `v` and returns that index to `selectSort` as `minI`. Then the values at position `i` and `minI` are exchanged, putting the next smallest value in the ith position.

In this strategy for each position in `v` every element in positions higher than the one currently being checked must be examined to find the index of the next smallest value. This means that if the array has $n$ elements, then in worst case we need to look at $n-1$ elements to find the index of the next smallest, so we say that we need approximately $n^2$ comparisons to put the array in order.

It is useful to note that there are many other sorting strategies, some more efficient that the bubble strategy or the selection strategy, and there are more advanced courses in which such strategies are discussed. Here our goal is to see a couple of sorting strategies and to challenge you to think of other possibilities.

## 11.3 exercises

**Exercise 11.1** *Write a program to search a file of 1000 random integers between 1 and 100 for the user's choice.*

**Exercise 11.2** *Write a program that searches a file of 1000 random integers between 1 and 100 for the user's choice, and stop when the first index with the required value is found.*

**Exercise 11.3** *Write a program to find all the indices where a particular integer is found.*

**Exercise 11.4** *Write a program that checks to see if a given random file of 1000 integers between 1 and 100 seems to reasonably random. HINT: We would expect to see approximately the same number of each integer.*

**Exercise 11.5** *Write a program that searches a file of names for the user's choice of a name.*

**Exercise 11.6** *Write 2 programs to sort a file of 1000 random integers between 1 and 100. One program should use the bubble strategy and one should use the selection strategy.*

**Exercise 11.7** *Write 2 programs to sort a file of names. One should use the bubble strategy and one should use the selection strategy.*

**Exercise 11.8** *Given a class that stores student records: name, major, and gpa, write a program that sorts the records by name. Write another program to sort them by major. Write another program to sort by gpa.*

# Bibliography

[1] ”JAVAPLT GROUP, R. U. Dr. java – a lightweight ide. http://www.drjava.org.

[2] KOLLING, M., AND BARNES, D. Bluej – the interactive java environment. http://www.bluej.org.