



## Chapter 2

# A Little Java

### Summing Up

The first chapter introduced a program development strategy of analysis, design, and implementation. Although you are encouraged to do some analysis and design before writing code, many problems encountered in the early chapters of this book will not require much effort to produce analysis and design deliverables. Analysis may simply be “Read the problem.” Design might end up as “I can picture the solution in my head.”

### Coming Up

This chapter will emphasize translating algorithms into programs using the Java programming language. Understanding how to translate a pseudocode algorithm into its programming language equivalent requires understanding the smallest pieces of a program and how to correctly gather them together to get things done. This chapter also examines how to deal with the input of numbers, the processing of numeric data, and output. An author-supplied class for reading input from the keyboard is also introduced. After studying this chapter, you will be able to

- use existing classes
- obtain data from the user and display information to the user
- declare and initialize primitive variables of types `double`, `int`, and `boolean`
- read numeric data from the keyboard
- construct an instance of a class and send messages
- evaluate and create arithmetic expressions and simple boolean expressions
- solve problems using the Input/Process/Output pattern

*Exercises and programming projects reinforce writing simple IPO programs with a text-based interface.*

---

## 2.1 The Java Programming Language — A Start

In the rawest definition, a Java **class** is a sequence of characters (text) stored as a file with a name that ends with `.java`. Each class consists of several elements such as a class heading (`public class class-name`). A class also contains methods—a collection of statements grouped together to perform a specific function (classes and methods are presented in detail in Chapter 4, “Classes and Interfaces”). Here is a general form for a Java class that has one method named `main`. A class with a **main method** can be run as a program.

### General Form 2.1: A simple Java program (only one class)

```
// Comments
import class-name
public class class-name
{
    public static void main( String[ ] args )
    {
        variable declarations and initializations
        class instance creations (constructing objects)
        messages and operations such as assignments
    }
}
```

A **general form** describes the **syntax**—the correct language—necessary to write classes and executable programs. This general form, like all others in this textbook, uses the following conventions:

1. Boldface elements must be written exactly as shown. This includes words such as **public static void main** and symbols such as `[ ]`, `( )`, and `.`
2. The programmer must supply the portions of a general form shown in italic. Italicized items are defined somewhere else or supplied by the programmer.

### An Example of a Syntactically Correct Java Class with One Method Named `main`

```
// Read a number and display that input value squared
import TextReader; // An author-supplied class in the working folder
import java.lang.System; // For System.out (automatically imported)

public class ReadItAndSquareIt // Class heading
{
    public static void main( String[] args ) // Method heading
    {
        double number; // Variable declaration
        double result = 0.0; // Variable initialization
        TextReader keyboard = new TextReader( ); // Class instance creation
    }
}
```

```

// I)input
System.out.print( "Enter a number: " ); // Message
number = keyboard.readDouble( );      // Message and assignment

// P)rocess (* means multiplication)
result = number * number;             // Assignment

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

```

### Dialogue

```

Enter a number: -12.0
-12.0 squared = 144.0

```

The first line is a comment documenting what the program does. The next line contains the word **import**, which allows a program to gain access to other classes. The class above has access to a class named `TextReader` for reading user input. You will often see a class name with a package name (a **package** is a collection of classes), where periods separate the folders containing the class. For example:

```

// Import one class from the java.io package
import java.io.BufferedReader;
// * means import all classes from the java.io package
import java.io.*;

```

The class above does not need the imports as shown. The important thing is that the file named `TextReader.java` is in the same folder as the Java program shown. Then, with or without `import TextReader;`, the program will compile. However, if `TextReader.java` is not in the same folder, the compiler will complain that `TextReader` is not known. So, to do input with the author-supplied **TextReader** class, you could simply place the file named `TextReader.java` into your working folder.<sup>1</sup> Here is one possible error message you would see if you did not get `TextReader.java` into the same folder as your program that tried to use it:

```

ReadItAndSquareIt.java:2: Class TextReader not found in import.
import TextReader; // An author-supplied class
    ^
1 error

```

### The other import

```

import java.lang.System; // For System.out

```

---

1. The file `TextReader.java` with class `TextReader` is stored on the disk that accompanies this textbook. This file must be copied into the same folder (directory) as the program that uses it.

is unnecessary because Java automatically imports all of the classes from `java.lang`. It's as if Java automatically writes this line for you at the top of each Java program:

```
import java.lang.*; // For all classes in java.lang: System, String ...
```

Java classes are organized into packages (approximately 72 at last count). Each package contains a set of related classes. For example, `java.net` has classes related to programming with the Internet, and `java.io` has a collection of classes for performing input and output. In this textbook, you will need only a few classes in a few packages to get things done. Many of these classes are in the `java.lang` package. The `import` is not necessary. Many programs will require one or more of the author-supplied classes, such as `TextReader` and `BankAccount`, which are stored in `TextReader.java` and `BankAccount.java`, respectively. It is recommended that you copy these files into your working folder and skip the optional `import`.

Many of the classes in the early part of this textbook can be completed without an `import`. Optional imports are shown in this chapter and the next one to remind you that you are using other classes to get things done. Later on, the unnecessary imports will not be written.

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

### General Form 2.2: The file-naming convention

*class-name.java*

The next line is a method heading that for now is best retyped exactly as shown:

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

The curly braces begin and end the `main` method. A method is a collection of executable statements and variables (both discussed later). When a class contains a `main` method, it can be run as a program. The program begins with the first statement in `main`. The `main` method above contains a variable declaration, a variable initialization, an object construction, and four messages, all of which are described later in this chapter.

This Java source code represents input to the Java compiler. A **compiler** is a program that translates source code into a level that is closer to the computer hardware. Along the way, the compiler generates error messages when it detects a violation of Java syntax rules. You will see errors detected as your compiler scans the source code of the programs you write.

## Tokens — The Smallest Pieces of a Program

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

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

These tokens are used to make up larger things. Knowing the types of tokens in Java should help you to

- more easily code syntactically correct messages
- better understand how to fix syntax errors detected by the compiler
- understand general forms

## Special Symbols

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

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

## Identifiers

A Java **identifier** is a wordlike token that represents a variety of things. For example, `String` is the name of a class for storing a string of characters. Here are some other identifiers that Java has already given meaning to:

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

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

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

**Valid Identifiers**

main	Vector	incomeTax	MAX_SIZE	\$Money\$
Maine	URL	employeeName	all_4_one	_balance
miSpel	String	A1	world_in_motion	my_balance

**Invalid Identifiers**

```

1A          // It begins with a digit
miles/Hour  // The / is unacceptable
first Name  // The blank space is unacceptable
pre-shrunk  // The operator - means subtraction
double      // It is a keyword

```

Remember the case sensitivity of Java. For example, many programs must have the identifier `main`. `MAIN` or `Main` won't do. Also, note that several conventions may be used for upper- and lowercase letters. Some programmers prefer to avoid uppercase letters; others prefer to use uppercase letters for each new word. The convention this textbook uses is the “camelBack” style for variables. This means each new word has an uppercase letter. For example, you will see `letterGrade` rather than `lettergrade`, `LetterGrade`, or `letter_grade`. All author-supplied classes follow the Java convention of beginning class names with an uppercase character: `String`, `Vector`, `Grid`, and `BankAccount`, for example.

**Keywords (Reserved Identifiers)**

A reserved identifier is an identifier that has been set aside for a specific purpose. It is a word whose meaning is fixed by the standard language definition, such as `double` and `int`. They follow the same rules as identifiers, but they cannot be used for any other purpose. Here is a partial list of Java reserved identifiers, which are also known as **keywords**.

**Java Keywords**

<code>boolean</code>	<code>default</code>	<code>for</code>	<code>new</code>
<code>break</code>	<code>do</code>	<code>if</code>	<code>private</code>
<code>case</code>	<code>double</code>	<code>import</code>	<code>public</code>
<code>catch</code>	<code>else</code>	<code>instanceOf</code>	<code>return</code>
<code>char</code>	<code>extends</code>	<code>int</code>	<code>void</code>
<code>class</code>	<code>float</code>	<code>long</code>	<code>while</code>

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

## Literals

A constant is a value that cannot be changed. As in mathematics, values such as 123 and 4.56 are constant values. When written inside a Java program, such constant values are also known as literal values, or simply **literals**.<sup>2</sup> For example, the Java compiler recognizes character literals as one character enclosed within single quotation marks. A `String` literal has zero or more characters enclosed within a pair of double quotation marks (special symbols) and finished on the same line.

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

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

### *The Six Types of Literals in Java*

Integer	Floating Point	String	Character	Boolean	Null
-2147483648	-1.0	"A"	'a'	true	null
-1	0.0	"Hello World"	'0'	false	
0	39.95	"\n new line"	'?'		
1	1.234e02	"1.23"	' '		
2147483647	-1e6	"The answer is: "	'7'		

## Comments

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

- Provide internal documentation to help one programmer read another's program—assuming those comments clarify the meaning of the program.
- Explain certain code fragments or the purpose of a variable.
- Indicate the programmer's name and the goal of the program.
- Describe a wide variety of program elements and other considerations.

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

```
/*
  A comment may
  extend over
  many lines
*/
```

---

2. Synonyms for *literal* include *exact*, *accurate*, and *plain*.

An alternate form for comments is to use `//` before the text. Such a comment may appear on a line by itself or at the end of a line.

```
// A complete Java program
public class DoNothing
{
    public static void main( String[] args )    // A method heading
    {
        // This program does nothing
    }
}
```

Within the context of the programs in this textbook, comments are most often written as one-line comments like `//Comment` rather than `/*Comment*/`. All code after `/*` is a comment until `*/` is encountered, so a large portion of the program could accidentally be turned into a comment by forgetting `*/` at the end! The one-line comments make it more difficult to accidentally “comment out” large sections of code. But many programmers prefer `/*` followed by `*/`.

Comments are added to help clarify and document the purpose of the source code. The goal is to make the program more understandable, easier to debug (correct errors), and easier to maintain (change when necessary). Programmers need comments to understand programs that may have been written days, weeks, months, years, or even decades ago.

Java has a third style of comments known as **javadoc comments**. Comments that begin with `/**` (note the second asterisk) and end with `*/` contain documentation and special tags that show up in Web pages (when you run the javadoc program).

```
/** Debit this account by withdrawalAmount if it is
 * no greater than this account's current balance.
 * @param withdrawalAmount The requested amount of money to withdraw.
 * @return true if 0 < withdrawalAmount <= the balance.
 */
public boolean withdraw( double withdrawalAmount )
```

Appendix A describes javadoc comments, tags such as `@param` and `@return`, and how to create hyperlinked Web pages with Java’s javadoc program. Javadoc comments document all of Java’s classes and interfaces. This allows you to have all of your classes documented in the same manner. This documentation can be read by a Web browser such as Netscape or Internet Explorer.

### Self-Check

- 2-1** How many tokens are there in the first program of section 2.1 (`ReadItAndSquareIt`)? *Note:* `System.out.println` has five tokens: two periods, which are special symbols, and three identifiers. Do not count comments.
- 2-2** Identify whether each of the following is a valid identifier or explain why it is not valid:

<b>-a</b> abc	<b>-l</b> H.P.
<b>-b</b> 123	<b>-m</b> \$Money\$Money\$Money
<b>-c</b> ABC	<b>-n</b> 55_mph
<b>-d</b> .	<b>-o</b> sales Tax
<b>-e</b> my Age	<b>-p</b> main
<b>-f</b> k/s	<b>-q</b> a
<b>-g</b> Abc!	<b>-r</b> _
<b>-h</b> identifier	<b>-s</b> _____
<b>-i</b> (identifier)	<b>-t</b> Mile/Hour
<b>-j</b> Double	<b>-u</b> student name
<b>-k</b> misspelled	<b>-v</b> TextReader

- 2-3** List two special symbols that are one character long.
- 2-4** List two special symbols that are two characters long.
- 2-5** List two identifiers that have already been defined by Java.
- 2-6** Create two programmer-defined identifiers.
- 2-7** Which of the tokens shown below are valid:

**-a** string literals?  
**-b** integer literals?  
**-c** floating-point literals?  
**-d** boolean literals?  
**-e** null literals?  
**-f** character literals?

```
1.0   34   'H'   "integer"   -123   1.0e+03   "H"   null
true   2,000   FALSE
```

- 2-8** Which of the following are valid Java comments?
- a** // Is this a comment?  
**-b** / / Is this a comment?  
**-c** /\* Is this a comment?  
**-d** /\* Is this a comment? \*/

---

## 2.2 Primitive Types

Java has two types of variables: primitive types and reference types. **Primitive variables** store a single value in a fixed amount of computer memory. The term “primitive” (simple) describes the eight different Java data types that store only one value. The primitive types are closely related to computer hardware. For example, an int value is stored in 32 bits (4 bytes) of memory. Those 32 bits represent a

simple positive or negative integer value. By contrast, a reference variable stores the address of (a reference to) an object.

Whereas the primitive types are part of Java for efficiency—their use helps make programs run faster—reference types allow for objects that are more intricate. Objects allow programmers to model real-world entities that are more intricate than numbers and single characters. Each object has a collection of related data and methods needed to fulfill the responsibilities of the class. Whereas primitive variables store only one value, a **reference variable** stores the address of an object. The object may store many primitive values and other references to other objects. The state of an object can be quite complex. However, the reference value is always the same type of value—an integer representing the address of the object. Here is summary of all data types in Java:

### *Java's Data Types*

#### **Primitive Types**

integers:

byte (8 bits)  
short (16 bits)  
int (32 bits)  
long (64 bits)

real numbers:

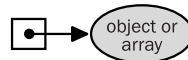
float (32 bits)  
double (64 bits)

others:

char (16 bits)  
boolean (true or false)

#### **Reference Types**

classes (Chapter 4)  
arrays (Chapter 8)



interfaces (Chapter 4)

Two of Java's primitive types for storing numbers—**int** and **double**—are used in virtually all programs. Their use is introduced by considering the following:

- how to declare and initialize variables
- how to output—display—the value of a variable
- how to change the value of a variable (assignment)
- how to input values through keyboard input to be stored into variables

## **Declarations and Initializations**

A primitive variable **declaration** brings into a program a named data value that can change while the program is running. An **initialization** allows the programmer to set the initial value of the variable. These variable names are used later when the programmer is interested in the current value of the variable or needs to change the value of that variable. Here are the general forms for declaring and initializing variables:

**General Form 2.3: Initializing (declaring a variable and giving it a value you want)**

```
data-type identifier;           // Declare one variable
data-type identifier = initial-value; // For primitive types like int and double
```

The `data-type` may be any of Java's primitive types, such as `double` to store numbers with a decimal point or `int` to store integers. Java's primitive types consist of `int`, `double`, `boolean`, `char`, `float`, and three integer types that store different ranges of integers: `byte`, `short`, and `long`. The following Java program declares one `int` and two `double` primitive variables:

```
// Initialize some numeric variables only. There is no input.
public class ShowSomePrimitiveTypes
{
    public static void main( String[] args )
    {
        int credits = 0;
        double qualityPoints; // Value is undefined
        double GPA = 0.0;    // ... nothing visible happens ...
    }
}
```

The primitive numeric types `int` and `double` can be initialized with `=` followed by the initial value. This satisfies all three attributes of a variable: (1) the name as specified by the identifier, (2) the available operations as specified by the type, and (3) the value as specified by the initial value after `=`. The following table summarizes the initial value of these variables:

<b>Variable Name</b>	<b>Value</b>
<code>credits</code>	0
<code>qualityPoints</code>	? // Unknown
<code>GPA</code>	0.0

**Output with `print` and `println`**

Programs communicate with users. Such communication is provided through—but is not limited to—keyboard **input** and screen **output**. This two-way communication is a critical component of many programs and is made possible by sending messages. A message is composed of several tokens that have been correctly grouped together to perform some operation. A message performs some well-defined responsibility while hiding many complexities. Some messages evaluate to some value that will be used by some other part of the program. Other messages just go off and perform some activity. Here are the general forms of two different Java messages. Both write information to the computer screen.

### General Form 2.4: Output

**System.out.print( *expression-1* + *expression-2*, ..., + *expression-n* );**

**System.out.println( *expression-1* + *expression-2*, ..., + *expression-n* );**

System.out is a variable that represents the computer screen. The expressions between parentheses will be displayed to the computer screen. *expression-1* through *expression-n* may take the form of variable names such as GPA or literals such as "Credits: " and 99.5. These expressions are separated with the concatenation operator + and enclosed within the set of parentheses. All of the values will be combined into one big string that is displayed on the computer screen. This expression, between the parentheses, is known as the **argument**. Finally, a semicolon (;) terminates the message.

The only difference between print and println is that println generates a new line so subsequent output begins at the beginning of a new line. Here are some valid output messages:

```
System.out.print( "Enter final exam: " );    // Use print to prompt
System.out.println( "Credits: " + credits );
System.out.println( );    // Cursor goes to beginning of next line
```

A println message with no arguments—nothing between the parentheses—generates a blank line. The above initializations and messages are now placed together in a new Java program that generates some rather meaningless output:

```
// Display some meaningless output that will make sense later
import java.lang.System;    // Or omit this since the import is automatic

public class ShowSomePrimitiveTypes
{
    public static void main( String[] args )
    {
        int credits = 12;
        double qualityPoints = 0.0;
        double GPA = 0.0;

        System.out.println( 99.5 );
        System.out.print( "Enter credits: " );
        System.out.println( " WHERE'S THE INPUT?" );
        System.out.println( );    // Generate a new line
        System.out.println( "Credits are " + credits + " for now." );
        System.out.println( 1 * 2.3 * 4 );
    }
}
```

### Output

99.5

Enter credits:  WHERE'S THE INPUT?

Credits are 12 for now.

9.2

### Self-Check

- 2-9** Write code to initialize two primitive numeric variables with an initial value of `-1.0`. Use any variable names you want.
- 2-10** Write a complete Java program that displays your name.

## Assignment

An **assignment** furnishes and/or modifies the value of a variable. The value of the expression to the right of `=` replaces whatever value was in the variable to the left of `=`.

### General Form 2.5: Assignment

*variable-name = expression;*

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

```
credits = 15;
qualityPoints = 45.67;
```

After the two assignments execute, the value of both variables is modified and the values can be shown like this:

<u>Variable</u>	<u>Value</u>
credits	15
qualityPoints	45.67

Assignments come with compatibility rules. For example, a `String` literal cannot be assigned to a numeric variable. A floating-point number cannot be stored in an `int`.

```
qualityPoints = "Noooooo, you can't do that"; // ERROR
credits = 16.5; // ERROR—Can't store a floating-point number in an int
```

The compiler will report an error at the attempted assignment.

### Self-Check

- 2-11** Which of the following are valid attempts at assignment, given the execution of these two statements?
- ```
double aDouble = 0.0;
int anInt = 0;
```

|           |                                 |           |                                       |
|-----------|---------------------------------|-----------|---------------------------------------|
| <b>-a</b> | <code>anInt = 1;</code>         | <b>-e</b> | <code>aDouble = 1;</code>             |
| <b>-b</b> | <code>anInt = 1.5;</code>       | <b>-f</b> | <code>aDouble = 1.5;</code>           |
| <b>-c</b> | <code>anInt = "1.5";</code>     | <b>-g</b> | <code>aDouble = "1.5";</code>         |
| <b>-d</b> | <code>anInt = anInt + 1;</code> | <b>-h</b> | <code>aDouble = aDouble + 1.5;</code> |

The following program uses a Java assignment to give specific values to variables that are initially only declared—not initialized. Notice that you can declare many variables when the identifiers are separated by commas.

```
// A few assignment statements
import java.lang.System;    // System is in the java.lang package

public class ShowSomePrimitiveTypes
{
    public static void main( String[] args )
    {
        int credits;
        double qualityPoints, GPA;    // A list of variables

        credits = 16;                // Assignment
        qualityPoints = 49.5;         // Assignment
        GPA = qualityPoints / credits; // Divide before assignment

        System.out.println( "Credits: " + credits );
        System.out.println( "Quality points: " + qualityPoints );
        System.out.println( "GPA: " + GPA );
    }
}
```

### Output

```
Credits: 16
Quality points: 49.5
GPA: 3.09375
```

Be wary of uninitialized variables. If you do not initialize a variable, it cannot be used unless it is changed later through assignment (which is discussed in the next section). In summary, to properly use variables in a program, all three characteristics must be considered:

1. A variable must be given a name in a declaration or initialization.
2. A variable must be declared as a specific primitive type.
3. A variable should be given a meaningful value through initialization, assignment, input, or some other means.

## Input

To make programs more general—for example, to find the GPA for any student—variables are often assigned values through keyboard input. This allows the user to

enter any data desired. There are several options for obtaining user input from the computer keyboard. Perhaps the simplest option is the use of an author-supplied Java class named `TextReader`. `TextReader` has a collection of methods that allow for easy input of numbers and strings.

Before you can use `TextReader` methods such as `readDouble`, your code must ask for an instance of the class. The following are examples of class instance creations. The result is that `keyboard` and `inFile` are reference variables that refer to different sources of input.

### **Creating an Instance of `TextReader` to Read Numeric Input (requires `TextReader.java`)**

```
// Construct an object named keyboard to read numbers from the user
TextReader keyboard = new TextReader( );
// Can use inFile to read input data from the disk file in.dat
TextReader inFile = new TextReader( "in.dat" );
```

This code is also referred to as **constructing an object**, because it invokes a special method called a “constructor,” a method that has the same name as the class (`TextReader` here).

In general, objects (instances of a class) are constructed with the keyword `new` followed by *class-name*( *optional-arguments* ).

### **General Form 2.6: Constructing objects (initial values are optional)**

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

This expression to the right of `=` evaluates to a reference value—the location of the object in memory. That reference value is then stored in the reference variable to the left of `=`. This is actually an assignment of a reference value to a reference variable. Whereas assignments to primitive variables store primitive values, assignments to reference values store the reference to an object. In other words, you could write the same object creation as follows:

```
TextReader keyboard; // Not initialized. You cannot get input yet!
keyboard = new TextReader( ); // Now you can get keyboard input
```

`TextReader` is an author-supplied Java class. It is stored in a file named `TextReader.java` on this textbook’s accompanying CD. Like other Java classes, `TextReader` has a collection of related methods. Each method performs some well-defined responsibility. This particular `TextReader` class has a collection of methods for reading text input either from the user at the keyboard or from a file on a disk. Through the object constructions above, several methods are now available to the `TextReader` objects referenced via the reference variables `keyboard` and `inFile`.

`TextReader` methods are used to get text input from the keyboard and convert that text, for example, 3.45 and 99, into numbers. Here are two messages that allow users to input numbers into a program:

### **Numeric Input (requires `TextReader.java`)**

```
keyboard.readInt( );           // Pause until user enters an integer
keyboard.readDouble( );       // This gets a floating-point number
```

Here is the general form of sending a message to an object so the object performs some well-defined responsibility. (*Note: `object-name` must already be constructed.*)

### **General Form 2.7: Sending a message to an object**

*object-name . message-name( argument-list )*

When a `readInt` or `readDouble` message is sent to a `TextReader` object, the method pauses program execution until the user enters the proper input value and then presses the Enter key. If the user enters the number correctly, the input string will be converted into the proper machine representation of the number.

These two methods are examples of expressions that evaluate to some value. Whereas a `readInt` message evaluates to a primitive `int` value, a `readDouble` message evaluates to a primitive floating-point value. Because `readInt` and `readDouble` return numeric values, they are often seen as part of assignment statements. These messages will be seen in text-based input and output programs (ones with no graphical user interface).

For example, the following two messages prompt the user to enter a number with the `print` method. The `readDouble` message then causes a pause until the user enters a number. At that point, `readDouble` converts the text typed by the user into a floating-point number. That number is then assigned to the variable named `qualityPoints`. All of this happens in one line of code.

```
System.out.print( "Enter quality points: " ); // Prompt the user
qualityPoints = keyboard.readDouble( );      // Assign the number entered
// qualityPoints is a number typed by the user at the keyboard
```

To use the `TextReader` class, you must copy the file `TextReader.java` into the same folder as your programs (unless your lab has this set up for you already). This gives you access to the `TextReader` class and the `readInt()` and `readDouble()` methods. A standard alternative, with no extra classes needed, is discussed in the Programming Tips section. This alternative form of input uses classes from the `java.io` package. It may be used instead of using `TextReader`.

The following program uses a `TextReader` object to obtain input for computing the GPA of any student. This program assumes that the file `TextReader.java` is in the same folder as the file `ShowSomePrimitiveTypes.java`.

```

// Read an int and a double.
// This program assumes TextReader.java is in the same folder.
// An unnecessary import (automatically done): import java.lang.System;

public class ShowSomePrimitiveTypes
{
    public static void main( String[] args )
    {
        int credits = 0;
        double qualityPoints = 0.0;
        double GPA = 0.0;
        TextReader keyboard = new TextReader( );

        System.out.print( "Enter credits: " );
        // User inputs an integer, which gets stored in credits
        credits = keyboard.readInt( );

        System.out.print( "Enter quality points: " );
        qualityPoints = keyboard.readDouble( );

        GPA = qualityPoints / credits;
        System.out.println( "Credits: " + credits );
        System.out.println( "Quality points: " + qualityPoints );
        System.out.println( "GPA: " + GPA );
    }
}

```

### Dialogue (remember that user input is shown in boldface italic)

```

Enter credits: 15
Enter quality points: 48.0
Credits: 15
Quality points: 48.0
GPA: 3.2

```

One of the nice things about using `TextReader` for input is that if the user accidentally enters an invalid number, the user is simply asked to enter a valid number.

### Dialogue (when the user does not enter valid numbers with `TextReader`)

```

Enter credits: 16.0
Invalid integer. Try again.
Bad again
Invalid integer. Try again.
Invalid integer. Try again.
16
Enter quality points: Invalid input entered on purpose
Invalid floating-point number. Try again.
Invalid floating-point number. Try again.
Invalid floating-point number. Try again.
Invalid floating-point number. Try again.
Invalid floating-point number. Try again.
49.5
Credits: 16
Quality points: 49.5
GPA: 3.09375

```

## 2.3 Arithmetic Expressions

Many of the problems in the early chapters of this textbook require you to write arithmetic expressions. Arithmetic expressions are made up of two components: operators and operands. An arithmetic operator is one of the Java special symbols `+`, `-`, `/`, or `*`. The operands of an arithmetic expression may be numeric variable names such as `qualityPoints`, numeric literals such as `5` and `0.25`, and method calls that evaluate to a numeric value such as `Math.sqrt(4.0)`. Assuming `aDouble` is a `double`, the following expression has operands `aDouble` and `4.5`. The operator is `+`.

```
aDouble + 4.5
```

Together, the operator and operands determine the value of the arithmetic expression.

The simplest arithmetic expression is a numeric literal or numeric variable name. Arithmetic expressions may also have two operands with one operator (see the table below).

### *Arithmetic Expressions*

| <b>An Expression May Be</b>          | <b>Example</b>                        |
|--------------------------------------|---------------------------------------|
| numeric variable                     | <code>double aDouble</code>           |
| numeric literal                      | <code>100</code> or <code>99.5</code> |
| expression <code>+</code> expression | <code>aDouble + 2.0</code>            |
| expression <code>-</code> expression | <code>aDouble - 2.0</code>            |
| expression <code>*</code> expression | <code>aDouble * 2.0</code>            |
| expression <code>/</code> expression | <code>aDouble / 2.0</code>            |
| <code>( expression )</code>          | <code>( aDouble + 2.0 )</code>        |

The previous definition of expression suggests more complex arithmetic expressions are possible.

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

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

### **Some Binary Arithmetic Operators**

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

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

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

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

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

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

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

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

The unary operator evaluates before the binary \* operator: 3.5 times negative 2.0 results in negative 7.0.

Arithmetic expressions usually have variable names as operands. When Java evaluates an expression with variables, the variable name is replaced by its value. Consider the following code:

```
double numOne = 1.0;
double numTwo = 2.0;
double numThree = 3.0;
double answer = numOne + numTwo * NumThree / 4.0;
```

The following simulation evaluates this arithmetic expression by first substituting the value for all variables and then evaluating each subexpression using the Java precedence rules:

```
answer = numOne + numTwo * NumThree / 4.0;
answer = 1.0 + 2.0 * 3.0 / 4.0 // Substitute values
answer = 1.0 + 6.0 / 4.0 // * has precedence over +
answer = 1.0 + 1.5 // / has precedence over +
answer = 2.5
```

### Self-Check

**2-12** Evaluate the following arithmetic expressions:

```
double x = 2.5;
```

```
double y = 3.0;
```

**-a**  $x * y + 3.0$

**-b**  $0.5 + x / 2.0$

**-c**  $1.0 + x * 3.0 / y$

**-d**  $1.5 * ( x - y )$

**-e**  $y + -x$

**-f**  $( x - 2.0 ) * ( y - 1.0 )$

## The `boolean` Type and Simple Boolean Expressions

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

| <b>Operator</b> | <b>Meaning</b>           |
|-----------------|--------------------------|
| <               | Less than                |
| >               | Greater than             |
| <=              | Less than or equal to    |
| >=              | Greater than or equal to |
| ==              | Equal to                 |
| !=              | Not equal to             |

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

| <b>Boolean Expression</b>    | <b>Result</b>      |
|------------------------------|--------------------|
| <code>double x = 4.0;</code> |                    |
| <code>x &lt; 5.0</code>      | <code>true</code>  |
| <code>x &gt; 5.0</code>      | <code>false</code> |
| <code>x &lt;= 5.0</code>     | <code>true</code>  |
| <code>5.0 == x</code>        | <code>false</code> |
| <code>x != 5.0</code>        | <code>true</code>  |

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

```
// Initialize three boolean variables to false
boolean ready = false;
boolean willing = false;
boolean able = false;
double credits = 28.5;
double hours = 9.5;

// Assign true or false to all three boolean variables
ready = hours >= 8.0;
willing = credits > 20.0;
able = credits <= 32.0;

System.out.println( "ready: " + ready );
System.out.println( "willing: " + willing );
System.out.println( "able: " + able );
```

```
// If all three booleans are true, the boolean expression is true
System.out.println( "All 3 true? " + (ready == willing == able) );
```

### Output

```
ready: true
willing: true
able: true
All 3 true? true
```

### Self-Check

**2-13** Which expressions evaluate to `true`, assuming that `j` and `k` are initialized like this:

```
int j = 4;
int k = 8;
```

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

---

## 2.4 Prompt and Input

The output and input operations are often used together to obtain values from the user of the program. The program informs the user what must be entered with an output message and then sends an input message to get values for the variables.

This happens so often that this activity can be considered to be a pattern. The

**Prompt and Input** pattern has two activities:

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

### Algorithmic Pattern 2.1

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

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

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

### Dialogue

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

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

### General Form 2.8: Prompt and Input

```
System.out.println( "prompt for the_number: " );
the_number = keyboard.readDouble( ); // Or keyboard.readInt( );
```

## Self-Check

**2-14** Write the value for GPA given each of the dialogues shown below:

```
public class SomeArithmetic
{
    public static void main( String[] args )
    {
        // 0. Declare some numeric variables
        double c1, c2, g1, g2, GPA;
        TextReader keyboard = new TextReader( );

        // 1. Input
        System.out.print( "Credits for course 1: " );
        c1 = keyboard.readDouble( );
        System.out.print( "Grade for course 1: " );
        g1 = keyboard.readDouble( );

        System.out.print( "Credits for course 2: " );
        c2 = keyboard.readDouble( );
        System.out.print( "Grade for course 2: " );
        g2 = keyboard.readDouble( );

        // 2. Process
        GPA = ( ( c1 * g1 ) + ( c2 * g2 ) ) / ( c1 + c2 );

        // 3. Output
        System.out.println( "GPA: " + GPA );
    }
}
```

### Dialogue 1

```
Credits for course 1: 2.0
Grade for course 1: 2.0
Credits for course 2: 3.0
Grade for course 2: 4.0
-a GPA: _____
```

**Dialogue 2**

```
Credits for course 1: 1.5
Grade for course 1: 4.0
Credits for course 2: 3.0
Grade for course 2: 3.0
-b GPA: _____
```

**Dialogue 3**

```
Credits for course 1: 0.5
Grade for course 1: 4.0
Credits for course 2: 3.0
Grade for course 2: 0.0
-c GPA: _____
```

---

## 2.5 Java’s Math Class

When the first programmers were programming the first electronic computers, phrases like this were often heard: “Hey, can I borrow that square root routine?” It got to the point where the code was printed and pinned to the wall so the code could be shared. Then someone got the idea to store an electronic version of the algorithm so people could call it up whenever it was needed. Therefore, functions were born. Functions are known in Java as “methods.”

Since then, many methods have been made part of programming language libraries. Some sets of methods relate to processing strings, others to graphical user interfaces, and still others to transferring information over networks. This section presents methods that help solve mathematical and scientific problems. The related methods are collected together by Java’s `Math` class. The Java **Math class** defines a large collection of mathematical and trigonometric functions (methods). Here are three:

```
Math.sqrt( number ) // Return the square root of number
Math.pow( x, y )     // Return x to the yth power
Math.sin( angle )   // Return the sine of angle radians
```

These `Math` methods are called by specifying the class name `Math`, followed by a dot, followed by the name of the method, followed by the appropriate number and type of arguments within parentheses. This type of call represents a major difference between the `Math` class and most other classes in Java. You do not need to construct a `Math` object to send messages (as was done for `TextReader`). The `Math` methods are for processing primitive types, not objects. When a method can be called by preceding the method name with the class name rather than a reference variable, the method is known as a **static method**. The keyword `static` will be part of the method. For example, `main` is a static method (other static methods, which

you will see in Chapter 5, include `getCurrencyInstance` of the `NumberFormat` class and `showMessageDialog` of the `JOptionPane` class).

`Math` methods typically require one argument and return a number. Here is one general form to call `Math` methods.

### ***General Form 2.9: Calling a method in the Math class***

**Math.** *method-name*( *argument(s)* )

The *method-name* is a previously declared identifier representing an operation that belongs to the `Math` class. The arguments represent a set of zero or more expressions separated by commas. In the following method call, the method name is `sqrt` (square root) and the argument is `81.0`:

```
Math.sqrt( 81.0 ) // Example method call that returns 9.0
```

Methods require zero, one, or even more arguments. Although most `Math` methods require exactly one argument, the `pow` method requires exactly two arguments. In the following message, the method name is `pow` (for power), the arguments are `base` and `power`, and the method call `pow( base, power )` is replaced with `basepower`, which returns `8.0`:

```
double base = 2.0;
double power = 3.0;
System.out.println( pow( base, power ) ); // Output: 8.0
```

Any argument used in a method call must be an expression from an acceptable type. For example, the method call `sqrt( "Bobbie" )` results in an error because the argument is a `String`, not a number. Here are some mathematical and trigonometric methods available to you. An example program that calls several of these methods follows.

### ***Some Math Methods***

| Method                      | Argument            | Return Type                    | Value Returned      | Example                       | Result  |
|-----------------------------|---------------------|--------------------------------|---------------------|-------------------------------|---------|
| <code>Math.cos(x)</code>    | <code>double</code> | <code>double</code>            | Cosine of x radians | <code>Math.cos(1.0)</code>    | 0.5403  |
| <code>Math.abs(x)</code>    | <code>double</code> | <code>double</code>            | Absolute value of x | <code>Math.abs(-1.5)</code>   | 1.5     |
| <code>Math.abs(i)</code>    | <code>int</code>    | <code>int</code>               | Absolute value of i | <code>Math.abs(-345)</code>   | 345     |
| <code>Math.pow(x, y)</code> | <code>double</code> | <code>double</code>            | x to the yth: $x^y$ | <code>Math.pow(2, 4)</code>   | 16.0    |
| <code>Math.round(x)</code>  | <code>double</code> | <code>long</code> <sup>3</sup> | Nearest integer     | <code>Math.round(4.56)</code> | 5       |
| <code>Math.sin(x)</code>    | <code>double</code> | <code>double</code>            | Sine of x radians   | <code>Math.sin(1.0)</code>    | 0.84147 |
| <code>Math.sqrt(x)</code>   | <code>double</code> | <code>double</code>            | Square root of x    | <code>Math.sqrt(4.0)</code>   | 2.0     |

---

3. The `round` method shown here takes a `double` and returns a `long`, which is an `int` with more significant digits. The `Math` class has another `round` method that takes a `float` and returns an `int` that would have to be called as in `int roundedVal = Math.round( 1.5F );`, where `1.5` is a `float` value, not a `double` value.

```
// Show the return value from calling Math methods.
// The Math class is automatically imported.
import java.lang.Math; // The Math class is in the package java.lang

public class ShowMathMethods
{
    public static void main( String[] args )
    {
        double x = -2.1;

        System.out.println( "x: " + x );
        System.out.println( "abs(x): " + Math.abs( x ) );
        System.out.println( "abs(10-27): " + Math.abs( 10 - 27 ) );
        System.out.println( "pow(x, 2.0): " + Math.pow( x, 2.0 ) );
        System.out.println( "round(x): " + Math.round( x ) );
        System.out.println( "round(3.5): " + Math.round( 3.5 ) );
        System.out.println( "round(9.9): " + Math.round( 9.9 ) );
    }
}
```

### Output

```
x: -2.1
abs(x): 2.1
abs(10-27): 17
pow(x, 2.0): 4.41
round(x): -2
round(3.5): 4
round(9.9): 10
```

Integer expressions may also be used as arguments to `Math` methods even though they are looking for a `double` argument. As with assignment, the integer value will be promoted to a `double`. So `Math.sqrt( 4 )` returns the same result as `Math.sqrt( 4.0 )` without error.

### Self-Check

- 2-15 Evaluate `Math.pow( 4.0, 3.0 )`.
- 2-16 Evaluate `Math.round( 3.49999 )`.
- 2-17 Evaluate `Math.abs( 123 - 125 )`.

---

## 2.6 int Arithmetic

The Java language provides several primitive **numeric types**. The two most often used are `double` and `int`. A variable declared as `int` can store a limited range of whole numbers—numbers with no fraction. Java `int` variables store integers in the range of -2,147,483,648 through 2,147,483,647 inclusive.

There are times when `int` is the correct choice over `double`. All `int` variables have operations similar to `double` (+, \*, -, =), but some differences do exist. For example, a fractional part cannot be stored in an `int`. In fact, you cannot assign a floating-point literal or `double` variable to an `int` variable. The compiler complains with an error.

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

The `/` operator has different meanings for `int` and `double` operands. Whereas the result of `3.0 / 4.0` is `0.75`, the result of `3 / 4` is `0`. Two integer operands with the `/` operator have an integer result—not a floating-point result. So what happens? An integer divided by an integer results in an integer. For example, the quotient obtained from dividing 3 by 4 is 0. This implies that the same operator (`/` in this case) can have a different meaning depending on the type of the operands.

The remainder operation—symbolized with the `%` operator—is also available for both `int` and `double`. For example, the result of `18 % 4` is the integer remainder after dividing 18 by 4, which is 2.

|                                                                                                          |                                                                                                             |                                                                                                                  |                   |
|----------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|-------------------|
| $9 / 2 \text{ is } 4$ $\begin{array}{r} \downarrow \\ 2 \overline{)9} \\ \underline{8} \\ 1 \end{array}$ | $10 / 2 \text{ is } 5$ $\begin{array}{r} \downarrow \\ 2 \overline{)10} \\ \underline{10} \\ 0 \end{array}$ | $403 / 4 \text{ is } 100$ $\begin{array}{r} \downarrow \\ 4 \overline{)403} \\ \underline{400} \\ 3 \end{array}$ | <b>Quotients</b>  |
| $9 \% 2 \text{ is } 1$ $\begin{array}{r} \uparrow \\ 2 \overline{)9} \\ \underline{8} \\ 1 \end{array}$  | $10 \% 2 \text{ is } 0$ $\begin{array}{r} \uparrow \\ 2 \overline{)10} \\ \underline{10} \\ 0 \end{array}$  | $403 \% 4 \text{ is } 3$ $\begin{array}{r} \uparrow \\ 4 \overline{)403} \\ \underline{400} \\ 3 \end{array}$    | <b>Remainders</b> |

**Integer arithmetic** is illustrated in the following program, which shows `%` and `/` operating on integer expressions and `/` operating on floating-point operands. In this example, the integer results describe whole hours and whole minutes rather than the fractional equivalent.

```
// Show minutes in two different ways
public class QuotientRemainder
{
    public static void main( String[] args )
    {
        System.out.println( "This program will show a given number" );
        System.out.println( "of minutes in two different ways." );
        System.out.println( );

        int totalMinutes, minutes, hours;
        double fractionalHour;
        TextReader keyboard = new TextReader( );
```

```

// 1. Input
System.out.print( "Enter total minutes: " );
totalMinutes = keyboard.readInt( );

// 2. Process
fractionalHour = totalMinutes / 60.0;
hours = totalMinutes / 60;
minutes = totalMinutes % 60;

// 3. Output
System.out.print( totalMinutes + " minutes can be rewritten as " );
System.out.println( fractionalHour + " hours or as" );
System.out.println( hours + " hours and " + minutes + " minutes" );
}
}

```

### Dialogue

This program will show a given number of minutes in two different ways.

```

Enter total minutes: 254
254 minutes can be rewritten as 4.23333 hours or as
4 hours and 14 minutes

```

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

### Self-Check

**2-18** What value is stored in nickel?

```

int change = 97;
int nickel = 0;
nickel = change % 25 % 10 / 5;

```

**2-19** What value is stored in nickel when change is initialized to:

|              |              |
|--------------|--------------|
| <b>-a</b> 4  | <b>-d</b> 15 |
| <b>-b</b> 5  | <b>-e</b> 49 |
| <b>-c</b> 10 | <b>-f</b> 0  |

## Mixing Integer and Floating-Point Operands

Whenever integer and floating-point values are on opposite sides of an arithmetic operator, the integer operand is promoted to its floating-point equivalent (3 becomes 3.0, for example). The expression then results in a floating-point number. The same rule applies when one operand is an int variable and the other a double variable. Here are a few examples of expression with only two operands:

```

public class IntDouble
{
    public static void main( String[] args )
    {
        int number = 9;
        double sum = 567.9;

        System.out.println( sum / number ); // Divide a double by an int
        System.out.println( number / 2 ); // Divide an int by an int
        System.out.println( number / 2.0 ); // Divide an int by a double
        System.out.println( 2.0 * number ); // Get a double: 18.0 not 18
    }
}

```

### Output

```

63.1
4
4.5
18.0

```

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

### Self-Check

**2-20** Evaluate the following expressions:

**-a**  $5 / 9$

**-b**  $5.0 / 9$

**-c**  $5 / 9.0$

**-d**  $2 + 4 * 6 / 3$

**-e**  $(2 + 4) * 6 / 3$

**-f**  $5 / 2$

**-g**  $7 / 2.5 * 3 / 4$

**-h**  $1 / 2.0 * 3$

**-i**  $5 / 9 * (50.0 - 32.0)$

**-j**  $5 / 9.0 * (50 - 32)$

---

## 2.7 Errors

There are several categories of errors encountered when programming:

**syntax errors**—errors that occur when compiling source code into byte code

**intent errors**—the program does what you typed, not what you intended (also known as “logic errors”)

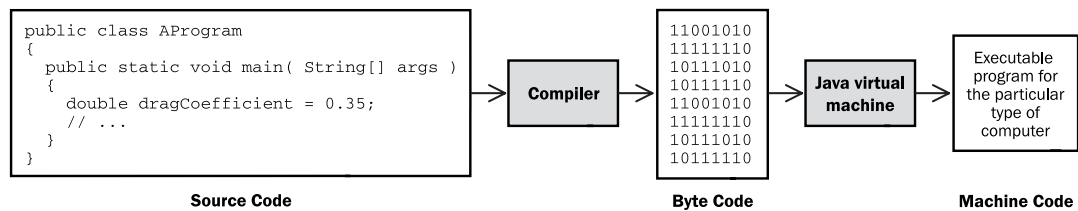
**exceptions**—errors that occur while the program is executing (also known as “runtime errors”)

When programming, you will be writing source code using the syntax for the Java programming language. This source code is translated into byte code by the

compiler. This byte code is stored in a `.class` file. The byte code is the same no matter what computer the program will run on.

For a Java program to actually run, another program called the **Java virtual machine** (JVM) translates the Java byte code into the instructions understood by the type of computer it runs on. The computer might be running Windows, Mac OS, Solaris Unix, or Linux, for example. Different computer systems have their own Java virtual machines. This is what allows the same Java Applet to be transported as a `.class` file around the Internet to be run on a variety of computers. Figure 2.1 shows the levels of translation necessary to get a computer-based solution using Java to get an executable program on almost any computer.

**Figure 2.1: From Source Code to a Program That Runs on Many Computers**



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

Errors may occur when you are compiling your source code or when the program is running on the computer. The easiest errors to detect and fix are those generated by the compiler. These are syntax errors that occur at compiletime.

## Syntax Errors Detected at Compiletime

A programming language requires strict adherence to its own set of formal syntax rules. Unfortunately, it is easy to violate these syntax rules while translating algorithms into their programming language equivalents. All it takes is a missing `{` or `;` to really foul things up. As you are writing your source code, you will often use the compiler to check the syntax of the code you wrote (see Figure 2.1). As the Java compiler translates source code into byte code so it can run on a computer, it also locates and reports as many errors as possible. If you have any syntax error, the byte code will not be generated. The program will not run. To get a running program, you need to fix all of your syntax errors.

A syntax error occurs when the compiler recognizes the violation of a syntax rule. The byte code cannot be created until all syntax errors have been removed from the program. The compiler can generate many strange-looking error mes-

sages as it reads your source code. Unfortunately, deciphering these syntax errors takes practice, patience, and a complete knowledge of the language. Of course, you will have to decipher these errors before you have complete knowledge of the language. Therefore, in an effort to improve this situation, here are some examples of common syntax errors and how they are corrected. *Note:* Your Java compiler will generate different error messages.

| <b>Error Detected by a Compiler</b> | <b>Incorrect Code</b>               | <b>Corrected Code</b>                                       |
|-------------------------------------|-------------------------------------|-------------------------------------------------------------|
| Splitting an identifier             | <code>int my Weight = 0;</code>     | <code>int <b>my</b>Weight = 0;</code>                       |
| Misspelling a keyword               | <code>integer sum = 0;</code>       | <code><b>int</b> sum = 0;</code>                            |
| Leaving off a semicolon             | <code>double x = 0.0</code>         | <code>double x = 0.0;</code>                                |
| Not closing a string literal        | <code>... print( "Hi );</code>      | <code>... print( "Hi" );</code>                             |
| Failing to declare a variable       | <code>testScore = 97.5;</code>      | <code><b>double testScore</b>;<br/>testScore = 97.5;</code> |
| Ignoring case sensitivity           | <code>double x;<br/>X = 5.0;</code> | <code>double x;<br/><b>x</b> = 5.0;</code>                  |
| Forgetting parentheses              | <code>keyboard.readDouble;</code>   | <code>keyboard.readDouble( );</code>                        |

Compilers generate many error messages. However, your source code is the source of these errors. You are the one that wrote the source code. Whenever your compiler appears to be nagging you, remember that the compiler is trying to help you correct your errors as much as possible.

*The compiler is your friend.*

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

```
public class Compile timeErrors
{
    public static void main( String[] args )
    {
        int pounds = 0;

        System.out.println( "Enter weight in pounds: " )①
        pounds = keyboard②.readInt( );
        System.out.print( "In the U.K. you weigh ③ );
        System.out.print( ④Pounds / 14 + " stone, " ⑤ pounds % 14 );
    }
}
```

Compilers generate some rather cryptic messages. Whereas one compiler reported only two errors with the source code above, there are actually these five errors.

- ❶ A semicolon (;) is missing
- ❷ keyboard was not constructed as a TextReader object
- ❸ A double quote (") is missing
- ❹ pounds was written as Pounds
- ❺ The extra expressions require a missing concatenation symbol (+)

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

```
// There are no syntax errors here

public class ErrorFree
{
    public static void main( String[] args )
    {
        int pounds = 0;
        TextReader keyboard = new TextReader( );

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

### Dialogue

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

Another type of error occurs when `String[] args` is omitted from the main method. When the program tries to run, it looks for a method named `main` with `( String[] identifier )`. If you forgot `String[] args` in the source code above, you would get the error shown after the program began to run.

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

These types of errors, which occur at runtime, are known as exceptions.

## Exceptions

After your program compiles with no syntax errors, you will get a `.class` file containing the byte code that can be run with the help of a Java virtual machine. The virtual machine translates the byte code into the instructions understood by

the computer (see Figure 2.1). The virtual machine can be invoked by issuing a Java command with the .class file name. For example, `java ErrorFree` will run the previous program on your computer.

However, when programs run, errors still may occur. Perhaps the user enters a string that is supposed to be a number. Or perhaps an arithmetic expression results in division by zero. Or there is an attempt to read a file from a disk, but there is no disk in the drive. Such exceptional events that occur while the program is running are known as exceptions.

One exception was shown above. The main method was valid, so the code compiled. However, when the program ran, Java's runtime environment looked for and could not find a main method with `String[] args`. The error could not be discovered until the user ran the program and Java began by looking for the place to begin the program. If Java cannot find a method with this line of code:

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

an exception occurs and the program terminates prematurely. Now consider another example of an exception that occurs while the program is running.

The output for the following program indicates that Java does not allow integer division by zero. The compiler does a lot, but it does not check the values of variables. Therefore, if at runtime a denominator in a division happens to be 0, an `ArithmeticException` occurs.

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

## **Output**

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at ExceptionExample.main(ExceptionExample.java:7)
```

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

## Intent Errors (Logic Errors)

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

```
// Find the average given the sum and the size of a set of numbers
public class IntentError
{
    public static void main( String[] args )
    {
        double sum = 0.0;
        double average = 0.0;
        int number = 0;
        TextReader keyboard = new TextReader( );

        // Input:
        System.out.print( "Enter sum: " );
        sum = keyboard.readDouble( );
        System.out.print( "Enter number: " );
        number = keyboard.readInt( );

        // Process
        average = number / sum;

        // Output
        System.out.println( "Average: " + average );
    }
}
```

The interactive dialogue may look like this:

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

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

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

### Self-Check

- 2-21** Assume a program is supposed to find an average given the sum and the size of a set of numbers and the following dialogue is generated. What clue reveals the presence of an intent error?

```
Enter sum: 100  
Enter number: 4  
Average: 0.04
```

- 2-22** Assuming the following code was used to generate the dialogue above, how is the intent error to be corrected?

```
System.out.print( "Enter sum: " );  
number = keyboard.readInt( );  
System.out.print( "Enter number: " );  
sum = keyboard.readDouble( );  
average = sum / number;
```

## When the Software Doesn't Match the Specification

Even when a process has been automated and delivered to the customer in working order as per the perceptions of the developers, there may still be errors. There have been many instances of software working, but not doing what it was supposed to do. This can happen when the software developers don't understand the customer's problem specification. Something could have been missed. Something could have been misinterpreted. The customer may not have expressed what was wanted.

A related error occurs when the customer specifies the problem incorrectly. This could be the case when the customer isn't sure what she or he wants. A trivial or critical omission in specification may occur, or the request may not be written clearly. In addition, the customer may change her or his mind after program development has begun.

For the most part, the Programming Projects in this textbook simply ask you to fulfill the problem specification. If there is something you don't understand, don't hesitate to ask questions. It is better to understand the problem and know what it is that you are trying to solve before getting to the design and implementation phases of program development. Unintentionally, the problem may have been incorrectly specified. Or it may have been incompletely specified. Both happen in the real world.

---

## Chapter Summary

- The smallest pieces of a program (tokens) can help you understand general forms and fix syntax errors. Java tokens include special symbols, identifiers, keywords (reserved identifiers), and literals.

- Java has two types of values: primitive values and reference values. A primitive variable stores one value in a fixed amount of bits. A reference variable stores the address of an object that usually has many values.
- Input is so frequently used that `TextReader` has been designed to make easy-to-use methods available. Numeric input without `TextReader` is complex and does not handle bad input very well. See the Programming Tips section for an alternative to `TextReader` that does not require the author-supplied file named `TextReader.java`.
- Arithmetic expressions are made up of operators, such as `+`, `-`, `*` (multiplication), and `/` (division). A binary arithmetic operator requires two operands, which may be numeric literals (1 or 2.3), numeric variables, or another arithmetic expression.
- Instances of the Prompt and Input pattern will occur in this textbook’s early programming projects. Use it whenever a program needs to get some input from the user.
- A message requires that an object be constructed. The message name is preceded by the object that will receive the message.
- Some classes have methods that do not need a reference variable. These are called “static methods.” They are called by preceding the method name with the class name. `Math.pow( 2, 4 )` is a call to a static method, for example.
- When `/` has two integer operands, the result is an integer. So `5 / 2` is 2.
- When `/` has at least one floating-point operand, the result is floating point: `5 / 2.0` is 2.5.
- The `%` operator returns the integer remainder of one integer operand divided by another. So `5 % 2` is 1. The `%` operator can also be used with floating-point operands. Then `%` returns the floating-point remainder. For example, `5.5 % 2.0` returns 1.5.
- Be careful in choosing `int` and `double`. Always use `double` to store numbers unless it makes sense that the variable will store only integers. For example, `credits` should be declared as a `double` since it is possible to have 0.5 credits in a course.
- This chapter ended with a discussion of some of the types of errors that occur during implementation. You will encounter errors. It is part of the process.
- Errors may be present because the problem statement was incorrect or incomplete.
- Intent errors can be the most difficult to fix—they are often difficult even to detect. Nothing is perfect, including programs that you pay for. They are often released with known errors (called “bugs” or “issues” in an effort to pretend they are not errors).
- Testing is important, but it does not prove the absence of errors. Testing can and does detect errors. Testing can build confidence that a program appears to work.

---




## Key Terms


|                        |                      |                          |
|------------------------|----------------------|--------------------------|
| abs                    | input                | println                  |
| argument               | int                  | Prompt and Input pattern |
| assignment             | integer arithmetic   | readDouble               |
| binary operator        | intent error         | readInt                  |
| boolean                | Java virtual machine | relational operator      |
| class                  | javadoc comments     | reference variable       |
| comment                | keyword              | round                    |
| compiler               | literal              | special symbol           |
| constructing an object | main method          | sqrt                     |
| declaration            | Math class           | static methods           |
| double                 | numeric type         | syntax                   |
| exception              | output               | syntax error             |
| general form           | package              | TextReader               |
| identifier             | primitive variable   | token                    |
| import                 | print                | unary operator           |
| initialization         |                      |                          |

---

## Exercises

The answers to exercises marked with  are available in Appendix C.

- List three operators that may be applied to a double.
-  List three operators that may be applied to any int.
-  Give one reason why a program should have comments.
- Describe how a programmer can change the value of a variable.
- List four types of Java tokens and give two examples of each.
-  Which of the following are valid identifiers?
 

|                     |                     |
|---------------------|---------------------|
| a. a-one            | g. 1_2_3            |
| b. R2D2             | h. A_B_C            |
| c. registered_voter | i. all right        |
| d. BEGIN            | j. "doubleVariable" |
| e. 1Header          | k. {Right}          |
| f. \$\$\$\$         | l. Mispelt          |
- Declare `totalPoints` as a variable capable of storing a floating-point number.
- Write code that sets the value of `totalPoints` to 100.0.
-  Write the entire dialogue generated by the following program when 5.2 and 6.3 are entered at the prompt. Make sure you write the user-supplied input as well as all program output, including the prompt.

```

public class Exercise9
{
    public static void main( String[] args )
    {
        double number = 0.0;
        double y = 0.0;
        double answer = 0.0;
        TextReader keyboard = new TextReader( );
        // 1. Input
        System.out.print( "Enter a number: " );
        number = keyboard.readDouble( );
        System.out.print( "Enter another number: " );
        y = keyboard.readDouble( );
        // 2. Process
        answer = number * ( 1.0 + y );
        // 3. Output
        System.out.println( "Answer: " + answer );
    }
}

```

10. Write a message that displays the value of a numeric variable `total`.



11. Given the following two variable initializations, either write the value that is stored in each variable or report the attempt as an error.

```

int anInt = 0;
double aNumber = 0.0;

```

- |                                |                                  |
|--------------------------------|----------------------------------|
| a. <code>anInt = 4;</code>     | d. <code>aNumber = 8;</code>     |
| b. <code>anInt = 4.5;</code>   | e. <code>aNumber = 8.9;</code>   |
| c. <code>anInt = "4.5";</code> | f. <code>aNumber = "8.9";</code> |



12. With paper and pencil, write a complete Java program that prompts for a number from 0.0 to 1.0 and stores this input value into the numeric variable named `relativeError`. Echo the input (output the input). The dialogue generated by your program should look like this:

```

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

```



13. A `println` message generates a new line on the computer screen. However, if you use `print` instead, the output appears on the same line. Write the output generated by the following programs:

```

public class A
{
    public static void main( String[] args )
    {
        System.out.print( "+++" );
        System.out.print( "+++" );
        System.out.print( "+++" );
    }
}

```

```

public class B
{
    public static void main( String[] args )
    {
        System.out.println( "+---+" );
        System.out.println( "+---+" );
        System.out.println( "+---+" );
    }
}

public class C
{
    public static void main( String[] args )
    {
        System.out.print( 1 );
        System.out.print( 2 );
        System.out.print( 3 );
    }
}

public class D
{
    public static void main( String[] args )
    {
        System.out.println( 1 );
        System.out.println( 2 );
        System.out.println( 3 );
    }
}

```

14. Assuming  $x$  is 5.0 and  $y$  is 7.0, evaluate the following expressions:
- |                                  |                                       |
|----------------------------------|---------------------------------------|
| a. $x / y$                       | g. $2.0 - x * y$                      |
| b. $y / y$                       | h. $( x * y ) / ( x + y )$            |
| c. $\text{Math.pow}( 2.0, 4.0 )$ | i. $\text{Math.round}( y + 0.3 - x )$ |
| d. $\text{Math.sqrt}( x - 1.0 )$ | j. $\text{Math.abs}( -23.4 )$         |
| e. $\text{Math.round}( -0.7 )$   | k. $\text{Math.round}( 0.6 )$         |
| f. $x > y$                       | l. $x \leq x$                         |
15. Predict the output generated by the following program:

```

public class Exercise15
{
    public static void main( String[] args )
    {
        double x = 1.2;
        double y = 3.4;

        System.out.println( x + y );
        System.out.println( x - y );
        System.out.println( x * y );
        System.out.println( x / y );
        System.out.println( x <= y );
        System.out.println( ( x + 5.0 ) < y );
    }
}

```

```

        System.out.println( x != y );
    }
}

```



16. Predict the output generated by the following program:

```

public class Exercise16
{
    public static void main( String[] args )
    {
        double x = 0.5;
        double y = 2.3;

        System.out.println( x * ( 1 + y ) );
        System.out.println( x / ( 1 + y ) );
        System.out.println( x / y );
    }
}

```

17. Write the complete dialogue (program output and user input) generated by the following program when the user enters each of the following input values for sale:

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

```

public class Exercise17
{
    public static void main( String[] args )
    {
        double sale = 0.0;
        double tax = 0.0;
        double total = 0.0;
        double TAX_RATE = 0.07;
        TextReader keyboard = new TextReader( );

        // I)input
        System.out.print( "Enter sale: " );
        sale = keyboard.readDouble( ); // User enters 10.00,
                                     // 12.34, and 100.00

        // P)rocess
        tax = sale * TAX_RATE;
        total = sale + tax;

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

```

18. Explain how to fix the syntax error in each of these lines:

a. `System.out.println( "Hello world" )`  
b. `System.println( "Hello world" );`

- c. `System.out.println( "Hello world );`  
 d. `System.out.println "Hello world";`



19. Explain the error in this program:

```
public class HasError
{
    public static void main( String[] args )
    {
        int anInt = 0;

        System.out.println( "Enter an integer" );
        anInt = keyboard.readDouble();
    }
}
```

20. Define the phrase *logic error*.



21. Does the following code always correctly assign the average of *x*, *y*, and *z* to *average*?

```
double average = x + y + z / 3.0;
```

22. What value is stored in *average* after this expression executes?

```
double average = ( 81 + 90 + 83 ) / 3;
```



23. Compute the value stored in *slope* given this assignment for the slope of a line:

```
slope = ( y2 - y1 ) / ( x2 - x1 );
```

| <u>x1</u> | <u>y1</u> | <u>x2</u> | <u>y2</u> | <u>slope</u> |
|-----------|-----------|-----------|-----------|--------------|
| 0.0       | 0.0       | 1.0       | 1.0       |              |
| 0.0       | 0.0       | -1.0      | 1.0       |              |
| 6.0       | 5.2       | 6.0       | -14.0     |              |

---

## Programming Tips

1. Use this outline for writing simple classes for your first projects.

```
// First-name last-name
// Comments describing what this program does

public class FileName
{
    public static void main( String[] args )
    {
        // Declare and/or initialize variables
        // Construct a TextReader object
```

```
        // Other messages and assignments
    }
}
```

## 2. Name the file the same as your class name.

The class name and the first part of the file name (before `.java`) must match exactly. Consider the following class (program) that has the name `FileNameDoesNotMatch` stored in a file named `NotHere.java`.

```
// File name: NotHere.java
public class FileNameDoesNotMatch
{
    public static void main( String[] args )
    {
        System.out.println( "Won't run" );
    }
}
```

This code compiles, but an attempt to run it resulted in this error message:

```
Can't find public class NotHere
```

To fix this, change either the file name or the class name.

## 3. Semicolons terminate. They do not belong at the end of every line.

Make sure you terminate declarations, assignments, and messages with `;`. However, do not place a semicolon after

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

## 4. Fix the first error first.

When you compile, you may get dozens of errors. Don't panic. Try to fix the very first error first. That may actually fix some of your other errors. Also note that sometimes fixing one error causes new errors. Don't be too surprised to find out that after fixing an error, the compiler generates errors that were previously undetected. It usually takes many tries to remove all errors.

## 5. Give meaningful values to your variables.

Although it is not always necessary to initialize `ints` and `doubles` at first, attempts to use the uninitialized variables later without a value will result in errors. Consider declaring variables when they are needed. Instead of this:

```
double credits, qualityPoints, GPA;
GPA = qualityPoints / credits;
```

which results in two Java syntax errors:

```
Variable qualityPoints may not have been initialized.
GPA = qualityPoints / credits;
    ^
```

```
Variable credits may not have been initialized.
GPA = qualityPoints / credits;
    ^
```

try declaring the variable when it is needed (instead of at the top of the method), like this:

```
double credits = keyboard.readDouble( );
// Prompt or whatever
double qualityPoints = keyboard.readDouble( );
// Prompt or whatever
double GPA = qualityPoints / credits;
```

## 6. Use intention-revealing names.

Although you may be tempted to use cryptic identifiers, such as `x` for credits, go ahead and make the extra keystrokes. Use a name that documents itself. All names should describe what they are intended for. This makes programs more readable to others. Perhaps even more importantly, this makes the program more readable to you. It will help you fix intent errors.

## 7. Integer arithmetic is different from calculators.

Integer division results in an integer. Therefore  $5 / 2$  is 2, not the 2.5 your brain and calculator feel are so right.

## 8. The `%` arithmetic operator causes confusion.

The expression `a % b` is the integer remainder after dividing `a` by `b`. Try these now:

```
99 % 50 = ____          101 % 2 = ____
99 % 50 % 25 = ____    102 % 2 = ____
4 % 99 = ____          103 % 2 = ____
```

## 9. Some imports are done automatically for you.

The classes in `java.lang` are used so often that you do not need to import them. This means you have the following three choices for a program that does nothing but print one line of output:

```
import java.lang.System; // Get one class
import java.lang.*;      // Get all classes in the package
// Do not write an import
```

Also, if the class is in a file in the same folder, you do not need to import it. For example, to get access to the `TextReader` class when `TextReader.java` is in the working folder, you have these two options:

```
import TextReader;
// Do not write an import
```

**10. Do more than is required.**

Consider doing more of the programming projects than you are assigned. Experience makes you a better programmer. You'll do better on the tests.

**11. To use the author-supplied class for reading input from the keyboard, copy `TextReader.java` into your working folder (directory).**

Some programming projects require author-supplied classes. These author-supplied classes are available to you if you have their files in the same folder as the program using the classes. You could use `TextReader.java` to complete the first programming projects. (It is available on the disk that accompanies this textbook.) Another option is given in the Programming Tip that follows. It requires more code, but you don't need the extra file.

**12. You can input with standard Java classes (classes that are always available).**

The standard Java's input classes provide another option for reading numbers. These classes are always available so there is no need to copy files into your working folder. However, this option comes with additional overhead for performing numeric input.

First, you need to write some extra code, which is shown in boldface in the program below. You also need to import several classes, such as `BufferedReader` and `InputStreamReader`. You must also declare that the main method throws `IOException`, which is another Java-supplied class in the `java.io` package. (See Chapter 9 for more on reading input with the classes in the `java.io` package.) These classes become available with the three imports in the program below.

```
// Read one number with the standard Java classes from java.io
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;    // Or write: import java.io.*;

public class UsingStandardInput
{
    public static void main( String[] args ) throws IOException
    {
        // Let the object named isr read characters from the keyboard.
        // System.in is a variable associated with the keyboard.
        InputStreamReader isr = new InputStreamReader( System.in );

        // Get an object with the convenient readLine method
        BufferedReader keyboard = new BufferedReader( isr );
        System.out.print( "Enter quality points: " );

        // Read in all user input up to the end of the line
        String numberAsString = keyboard.readLine( );
    }
}
```

```

    // Convert user input to a number or else throw an exception
    double qualityPoints = Double.parseDouble( numberAsString );
    System.out.println( "You entered " + qualityPoints );
}
}

```

The object referenced by `keyboard` has a `readLine` that reads a string of characters from the user. This string of characters must then be converted into a number by `Double`'s `parseDouble` method. The method requires a `String` argument that hopefully represents a number; "1.234", for example. The `parseDouble` message results in a `double` that is then assigned to the `double` variable `qualityPoints`.

### **Dialogue 1 (valid numeric input)**

```

Enter quality points: 34.5
You entered 34.5

```

If the user enters an invalid number, this particular program will terminate early.

### **Dialogue 2 (invalid numeric input)**

```

Enter quality points: 34..5
Exception in thread "main" java.lang.NumberFormatException:
at java.lang.Float.parseFloat(Float.java:188)
at java.lang.Double.parseDouble(Double.java:188)
at UsingKeyboard.main(UsingStandardInput.java:24)

```

One advantage to using this option is that you do not need the author-supplied `TextReader` class. But it is easy to copy it into the folder with your other files. If your instructor insists that you do not use `TextReader`, use the code above for reading doubles. To read integers, use `Integer.parseInt` rather than `Double.parseDouble`. Both methods require a `String` argument.

```

System.out.print( "Enter an int: " );
int anInteger = Integer.parseInt( standardInput.readLine( ) );

```

---

## **Programming Projects**

*Note:* This first programming project is provided as an example that could be followed to complete other projects. It takes you from start to finish. It is typical of the 15 programming projects in this chapter. Each project in this chapter has the following elements:

- a project number (2Z, for example) and a title (Rounding to a Given Number of Decimal Places, for example)
- a problem specification

- zero, one, two, or more sample dialogues, often asking you to reproduce the dialogue

## 2Z Rounding to a Given Number of Decimal Places

Write a program that rounds a number to a specific number of decimal places. For example, 3.4589 rounded to two decimal places should be 3.46 and -3.4589 rounded to one decimal place should be -3.5. One dialogue must look exactly like this:

```
Enter number to round: 3.4567
Enter number of decimal places: 2
Rounded number: 3.46
```

The following is a sampling of activities based on the programming process covered in Chapter 1.

## Analysis

You could begin with these analysis activities:

1. Read and understand the problem.
2. Decide what variable names represent the answer—the output.
3. Decide what variable names the user must enter to get the answer—the input.
4. Create a document that summarizes your analysis. This can be used as input to the design phase.

| <b>Problem</b>             | <b>Data Name</b> | <b>Input or Output</b> | <b>Sample Problem</b> |
|----------------------------|------------------|------------------------|-----------------------|
| Round to any decimal place | number           | Input                  | 3.45678               |
|                            | decimals         | Input                  | 3                     |
|                            | result           | Output                 | 3.457                 |

## Design

One suggested design activity is the development of an algorithm. Many algorithms for Chapter 2 projects can be guided by the Input/Process/Output pattern. This will help place the appropriate activities in the proper order. Add two instances of the Prompt then Input pattern, and an algorithm might look like this:

1. Prompt for the number to round (call it `number`).
2. Input `number`.
3. Prompt for the number of decimal places (call it `decimals`).
4. Input `decimals`.
5. Round `number` to `decimals` decimal places and store in `result`.
6. Display `result`.

Steps 1, 2, 3, 4, and 6 are easy. They can be implemented as input and/or output statements. However, the details of step 5, “Round number to decimals decimal places,” are not present. Step 5 needs refinement. Now the focus can shift to the more difficult process of rounding number to decimals decimal places. A solution is a bit tricky, so the method is provided. Step 5 will now be replaced with steps 5, 6, and 7.

To round number to decimals decimal places, first multiply number by 10 to the decimals power (stored in result below). Then round result to the nearest integer. Finally, divide result by 10 to the decimals power. The refined algorithm now looks like this:

1. Prompt for the number to round (call it number).
2. Input number.
3. Prompt for the number of decimal places (call it decimals).
4. Input decimals.
5. Let result become number multiplied by  $10^{\text{decimals}}$ .
6. Let result become the round of result.
7. Let result become result divided by  $10^{\text{decimals}}$ .
8. Display result.

Perform an algorithm walkthrough to simulate what will happen when number is 3.4567 and the result is to be rounded to two decimal places.

### Round 3.4567 to Two Decimal Places

$$\begin{aligned} \text{result} &= \text{number} * 10^{\text{decimals}} &&= 3.4567 * 10^2 &&= 345.67 \\ \text{result} &= \text{round}(\text{result}) &&= 345.67 + 0.5 &&= 346 \\ \text{result} &= \text{result} / 10^{\text{decimals}} &&= 346 / 10^2 &&= \mathbf{3.46} \end{aligned}$$

It helps to have more than one sample problem. Therefore, walk through the same algorithm with a different example problem, rounding 9.99 to one decimal place. You shouldn't need this elaborate table. You could just walk through your algorithm with different input values.

| Activity                                                           | number | decimals | result |
|--------------------------------------------------------------------|--------|----------|--------|
| 1. Prompt for the number to round.                                 | ?      | ?        | ?      |
| 2. Input number.                                                   | 9.99   | ?        | ?      |
| 3. Prompt for the number of decimal places.                        | 9.99   | ?        | ?      |
| 4. Input decimals.                                                 | 9.99   | 1        | ?      |
| 5. Let result become number multiplied by $10^{\text{decimals}}$ . | 9.99   | 1        | 99.9   |
| 6. Let result become the round of result.                          | 9.99   | 1        | 100.0  |
| 7. Let result become result divided by $10^{\text{decimals}}$ .    | 9.99   | 1        | 10.0   |
| 8. Display result.                                                 | 9.99   | 1        | 10.0   |

Before writing the program, come up with a few more sample problems and write the predicted result. Try a negative number. Try zero decimal places.

| <b>number</b> | <b>decimals</b> | <b>result</b> |
|---------------|-----------------|---------------|
| 0.004999      | 2               | 0.00          |
| 3.151         | 1               | 3.2           |
| -1.6          | 0               | -2.0          |

Now you can translate your algorithm into Java. This could be done with paper and pencil before going to the lab. This activity can help you do better on tests.

Next, type in the Java source code. This is the translation of your algorithm. You may need some instruction on using your specific Java programming environment. Compile repeatedly until all syntax errors are removed. Once the program compiles, run your program with different inputs. Test that your program works by using the same input values as above to generate some dialogues. For each set of inputs, compare your program results with your expected results.

## Implementation

This complete Java source code program is a translation of the previous algorithm. The algorithm step numbers are kept as comments to show how each step was translated. Output from two program runs was copied from the console and pasted at the end of the file. The technique on how to do this varies between computer systems.

```
// Programmer: Rick Mercer
// Project 2Z: Round any number to any number of decimal places
// Due Date: 11/14/03

import java.lang.System; // import not required
import TextReader; // TextReader.java must be in the same directory

public class RoundingNumbers
{
    public static void main( String[] args )
    {
        // Declare variables identified during analysis
        double number = 0.0;
        double decimals = 0.0;
        double result = 0.0;
        TextReader keyboard = new TextReader( );

        // Java translation of algorithm           Algorithm step number:
        // I)input
        System.out.print( "Enter number to round: " ); // 1.
        number = keyboard.readDouble( ); // 2.

        System.out.print( "Enter number of decimal places: " ); // 3.
        decimals = keyboard.readInt( ); // 4.
```

```

// P)rocess (Round number to decimals decimal places)
result = number * Math.pow( 10, decimals );           // 5.
result = Math.round( result );                       // 6.
result = result / Math.pow( 10, decimals );           // 7.

// O)utput
System.out.println( "Rounded number: " + result );    // 8.
}
}
/* Additional documentation: Two copied and pasted logs of execution. This
   may not be necessary, but it helps show the behavior of this program.

Enter number to round: 3.4567
Enter number of decimal places: 2
Rounded number: 3.46

Enter number to round: -1.6
Enter number of decimal places: 0
Rounded number: -2.0
*/

```

## 2A Simple Arithmetic

Implement and test a Java program that solves the problem specified in Exercise 1A, “Simple Arithmetic.” The problem: For any two numeric inputs, *a* and *b*, compute the product (*a* \* *b*) and the sum (*a* + *b*). Then display the difference between the product and the sum. Your dialogue must look exactly like this (except your input will not be in boldface italic):

```

Enter a number: 5.0
Enter a number: 10.0
Product = 50.0
Sum = 15.0
Product - sum = 35.0

```

## 2B Simple Average

Implement and test a Java program that solves the problem specified in analysis/design exercise 1B, “Simple Average.” The problem: Find the average of three tests of equal weight. One dialogue must look exactly like this (except your input will not be in boldface italic):

```

Enter test 1: 90.0
Enter test 2: 80.0
Enter test 3: 70.0
Average = 80.0

```

## 2C Weighted Average

Implement and test a Java program that solves the problem specified in analysis/design exercise 1C, “Weighted Average.” The problem: Determine a course grade using this weighted scale:

Quiz average    20%  
Midterm        20%  
Lab grade       35%  
Final exam     25%

One dialogue must look exactly like this (except your input will not be in boldface italic):

```
Enter Quiz Average: 90.0  
Enter Midterm: 80.0  
Enter Lab Grade: 100.0  
Enter Final Exam: 70.0  
Course Average = 86.5
```

## 2D Wholesale Cost

Implement and test a Java program that solves the problem specified in analysis/design exercise 1D, “Wholesale Cost.” The problem: You happen to know that a store has a 25% markup on compact-disc (CD) players. If the retail price (what you pay) of a CD player is \$189.98, how much did the store pay for that item (the wholesale price)? In general, what is the wholesale price for any item given its retail price and markup? Analyze the problem and design an algorithm that computes the wholesale price for any given retail price and any given markup. *Clue:* If you can’t determine the equation, use this formula and some algebra to solve for wholesale price:

$$\text{retail price} = \text{wholesale price} * (1 + \text{markup})$$

Your dialogues must look exactly like this (except your input will not be in boldface italic):

```
Enter the retail price: 255.00  
Enter the markup percentage: 50  
Wholesale price = 170.0  
  
Enter the retail price: 200.00  
Enter the markup percentage: 100  
Wholesale price = 100.0
```

## 2E Grade Point Average

Implement and test a Java program that solves the problem specified in analysis/design exercise 1E, “Grade Point Average.” The problem: Compute a student’s cumulative grade point average (GPA) for three courses. Credits range from 0.5 to

15.0. Grades can be 0.0, 1.0, 2.0, 3.0, or 4.0. One dialogue must look exactly like this (except your input will not be in boldface italic):

```
Credits for course 1: 2.0
  Grade for course 1: 2.0
Credits for course 2: 3.0
  Grade for course 2: 4.0
Credits for course 3: 3.0
  Grade for course 3: 4.0
GPA: 3.5
```

## 2F F to C

Use the following formula, which converts Fahrenheit (F) temperatures to Celsius (C):

$$C = 5/9(F-32)$$

Write a Java program that inputs a Fahrenheit temperature and outputs the Celsius equivalent. Your dialogues must look exactly like these two dialogues when 212.0 and 98.6 are entered for F (except your input will not be in boldface italic):

```
Enter Fahrenheit temperature: 212.0
212.0 Fahrenheit is 100.0 Celsius

Enter Fahrenheit temperature: 98.6
98.6 Fahrenheit is 37.0 Celsius
```

## 2G C to F

Use algebra and the formula of the preceding programming project to convert degrees Celsius (C) to degrees Fahrenheit (F). Write a Java program that inputs any Celsius temperature and displays the Fahrenheit equivalent. Your dialogues must look exactly like this when the user enters -40 and 37 for C (except your input will not be in boldface italic):

```
Enter C: -40.0
-40.0 C is -40.0 F

Enter C: 37.0
37.0 C is 98.6 F
```

## 2H Seconds

Write a program that reads a value in seconds and displays the number of hours, minutes, and seconds represented by the input. Your dialogues must look exactly like this (except your input will not be in boldface italic):

```
Enter seconds: 32123
8:55:23

Enter seconds: 61
0:1:1
```

## 2I U.S. Minimum Coins

Write a Java program that prompts for an integer that represents the amount of change (in cents) to be handed back to a customer in the United States. Display the minimum number of half dollars, quarters, dimes, nickels, and pennies that make the correct change. *Hint:* With increasingly longer expressions, you could use `/` and `%` to evaluate the amount of each coin. Or you could calculate the total number of coins with `/` and the remaining change with `%`. Verify that your program works correctly by running it with a variety of input. Your dialogues must look exactly like this (except your input will not be in boldface italic):

|                                 |                                 |
|---------------------------------|---------------------------------|
| Enter change [0..99]: <b>83</b> | Enter change [0..99]: <b>14</b> |
| Half(ves) : 1                   | Half(ves) : 0                   |
| Quarter(s) : 1                  | Quarter(s) : 0                  |
| Dime(s) : 0                     | Dime(s) : 1                     |
| Nickel(s) : 1                   | Nickel(s) : 0                   |
| Penny(ies) : 3                  | Penny(ies) : 4                  |

## 2J U.K. Minimum Coins

Write a Java program that prompts for an integer that represents the amount of change in pence to be handed back to a customer in the United Kingdom. Display the minimum number of coins that make the correct change. The available coins are (p represents pence) 1p, 2p, 5p, 10p, 20p, 50p, and 100p (the one-pound coin). Verify that your program works correctly by running it with a variety of input. Your dialogues must look exactly like this (except your input will not be in boldface italic):

|                          |                         |
|--------------------------|-------------------------|
| Enter change: <b>298</b> | Enter change: <b>93</b> |
| 100p: 2                  | 100p: 0                 |
| 50p: 1                   | 50p: 1                  |
| 20p: 2                   | 20p: 2                  |
| 10p: 0                   | 10p: 0                  |
| 5p: 1                    | 5p: 0                   |
| 2p: 1                    | 2p: 1                   |
| 1p: 1                    | 1p: 1                   |

## 2K Circle

Write a Java program that reads a value for the radius of a circle and then outputs the diameter, circumference, and area of the circle. Use the `Math.pow` method to compute the area.

- `diameter = 2 * radius`
- `circumference = pi * diameter`
- `area = pi * radius2`

*Note:* Use the variable `Math.PI`, which is as close to `p` as your computer allows. For example, this message generates the output shown in the comment:

```
System.out.println( Math.PI ); // 3.141592653589793
```

Run your program with radius = 1.0. Verify that your values for circumference and area match the preceding dialogue. After this, run your program with the input radii of 2.0 and 2.5 and verify that the output is what you expect. Your program must generate a dialogue that looks exactly like this when the input is 1.0 (except your input will not be in boldface italic):

```
Enter Radius: 1.0
Diameter: 2.0
Circumference: 6.283185307179586
Area: 3.141592653589793
```

## 2L More Rounding

Write a program that asks the user for a number and displays that number rounded to zero, one, two, and three decimal places. Your program must generate a dialogue that looks exactly like this when the input is 3.4567 (except your input will not be in boldface italic):

```
Enter the number to round: 3.4567
3.4567 rounded to 0 decimals = 3.0
3.4567 rounded to 1 decimal = 3.5
3.4567 rounded to 2 decimals = 3.46
3.4567 rounded to 3 decimals = 3.457
```

## 2M Range

Write a program that determines the range of a projectile using this formula:

$$\text{range} = \sin(2 * \text{angle}) * \text{velocity}^2 / \text{gravity}$$

where *angle* is the angle of the projectile's path (in radians), *velocity* is the initial velocity of the projectile (in meters per second), and *gravity* is the acceleration at 9.8 meters per second per second (a constant).

The take-off angle must be input in degrees. Therefore, you must convert this angle to its radian equivalent. This is necessary because the trigonometric method `Math.sin` assumes that the argument is an angle expressed in radians. An angle in degrees can be converted to radians by multiplying the number of degrees by `pi/180` where `pi` » 3.14159. For example,  $45^\circ = 45 * 3.14159 / 180$ , or 0.7853975 radians. In this problem, use the constant in the `Math` class named `Math.PI` for the value of `pi`. `Math.PI` is the closest value to `pi` that your computer can support.

Your program must generate a dialogue that looks exactly like this when the input is 45.0 degrees and the initial velocity is 100 meters per second (except your input will not be in boldface italic). Round your answer to two decimal places.

```
Takeoff Angle (in degrees): 45.0
Initial Velocity (meters per second): 100.0
Range = 1020.41 meters
```

## 2N Departure Times

Write a Java program that reads in two different train departure times (where 0 is midnight, 0700 is 7:00 a.m., 13:14 is 14 minutes past 1:00 p.m., and 2200 is 10 p.m.) and displays the difference between the two times in hours and minutes. The program should work even if the first departure time input is later in the day than the second departure time input. Assume both times are on the same date and that both times are valid. For example, 1099 is not a valid time because the last two digits are minutes, which should be in the range of 00 through 59. 2401 is not valid because the hours (the first two digits) must be in the range of 0 through 23 inclusive. Verify that your program works correctly by running it with a variety of valid input. Here are three sample dialogues with the correct results:

```
Train A departs at: 1255
Train B departs at: 1305
0 hours and 10 minutes
```

```
Train A departs at: 2350
Train B departs at: 0055
22 hours and 55 minutes
```

```
Train A departs at: 0730
Train B departs at: 0845
1 hours and 15 minutes
```

---

## Answers to Self-Check Questions

- 2-1 68 plus or minus a few. It is easy to miscount tokens. For example, comments are not tokens and this is only one token: "A very long string with new, int, double, { }, +, -, /, && \* "
- 2-2
- |                                 |                                          |
|---------------------------------|------------------------------------------|
| -a VALID                        | -l Periods (.) are not allowed.          |
| -b I can't start an identifier. | -m VALID                                 |
| -c VALID                        | -n Can't start identifiers with a digit. |
| -d . is a special symbol.       | -o A space is not allowed.               |
| -e A space is not allowed.      | -p VALID                                 |
| -f / is not allowed.            | -q VALID                                 |
| -g ! is not allowed.            | -r VALID (but not meaningful).           |
| -h VALID                        | -s VALID (but not meaningful).           |
| -i ( ) are not allowed.         | -t / is not allowed.                     |
| -j VALID (double is not)        | -u A space is not allowed.               |
| -k VALID                        | -v VALID                                 |
- 2-3 + - Also , : ; ! ( ) = { }
- 2-4 <= >= Also != == < >=
- 2-5 String System Also readLine println
- 2-6 thisIsOne and anotherOne

- 2-7 -a string literals: "H" and "integer"                      -d boolean literals: true  
 -b integer literals: 234 and -123                              -e null literals: null  
 -c floating-point literals: 1.0 and 1.0e+03                    -f character literals: 'H'
- 2-8 a and d only
- 2-9 double aNumber = -1.0;  
 double anotherNumber = -1.0;
- 2-10 public class YourName  
 {  
     public static void main( String[] args )  
     {  
         System.out.println( "Kim Davies" );  
     }  
 }
- 2-11 a, d, e, f, and h are valid.  
 b Attempts to store a floating-point literal into an int variable.  
 c Attempts to store a string literal into an int variable.  
 g Attempts to store a string literal into a double variable.
- 2-12 -a 10.5                      -d -0.75  
 -b 1.75                        -e 0.5  
 -c 3.5                         -f 1.0
- 2-13 -a true                      -e true  
 -b false                       -f true  
 -c false                       -g false  
 -d true                        -h true
- 2-14 Dialogue 1:                      Dialogue 2:                      Dialogue 3:  
 -a **3.2** 16.0 / 5.0                    -b **3.33** 15.0 / 4.5                    -c **0.57** 2.0 / 3.5
- 2-15 64.0
- 2-16 3.0
- 2-17 2
- 2-18 0
- 2-19 -a 0                      -d 1  
 -b 1                        -e 0  
 -c 0                        -f 0
- 2-20 -a 0                      -f 2  
 -b 0.55556                  -g 2.1  
 -c 0.55556                  -h 1.5  
 -d 10                       -i 0.0  
 -e 12                        -j 10.0
- 2-21 The average is wrong.
- 2-22 Change the prompts. "Enter sum: " becomes "Enter number: " and "Enter number: " becomes "Enter sum: ".