CHAPTER 3

SYNTAX, ERRORS, AND DEBUGGING

OBJECTIVES

Upon completion of this chapter, you should be able to:

- Construct and use numeric and string literals.
- Name and use variables and constants.
- Create arithmetic expressions.
- Understand the precedence of different arithmetic operators.
- Concatenate two strings or a number and a string.
- Know how and when to use comments in a program.
- Tell the difference between syntax errors, run-time errors, and logic errors.
- Insert output statements to debug a program.
- Understand the difference between Cartesian coordinates and screen coordinates.
- Work with color and text properties.

Estimated Time: 4.0 hours

VOCABULARY

Arithmetic expression
Comments
Coordinate system
Exception
Graphics context
Literal
Logic error
Origin
Package
Pseudocode
Reserved words
Run-time error
Screen coordinate system
Semantics
Syntax
Virus

To use a programming language, one must become familiar with its vocabulary and the rules for forming grammatically correct statements. You must also know how to construct meaningful statements and statements that express the programmer's intent. Errors of form, meaning, and intent are possible, so finally one must know how to detect these errors and correct them. This chapter discusses the basic elements of the Java language in detail and explores how to find and correct errors in programs.

3.1 Language Elements

Before writing code in any programming language, we need to be aware of some basic language elements. Every natural language, such as English, Japanese, and German, has its own vocabulary, syntax, and semantics. Programming languages also have these three elements.

Vocabulary is the set of all of the words and symbols in the language. Table 3-1 illustrates some examples taken from Java.

TABLE 3-1
Some Java vocabulary

TYPE OF ELEMENT	EXAMPLES
arithmetic operators	`+ - * /`
assignment operator	`=`
numeric literals	`5.73 9`
programmer defined variable names	`fahrenheit celsius`

Syntax consists of the rules for combining words into sentences, or *statements*, as they are more usually called in programming languages. Following are two typical syntax rules in Java:

1. In an expression, the arithmetic operators for multiply and divide must not be adjacent. Thus,

```
(f - 32) * / 9
```

   is invalid.

2. In an expression, left and right parentheses must occur in matching pairs. Thus,

```
)f - 32( * 5 / 9
```

   and

```
f - 32) * 5 / 9
```

   are both invalid.

Semantics define the rules for interpreting the meaning of statements. For example, the expression

```
(f - 32.0) * 5.0 / 9.0
```

means "go into the parentheses first, subtract `32.0` from the variable quantity indicated by `f`, then multiply the result by `5.0`, and finally divide the whole thing by `9.0`."

Programming versus Natural Languages

Despite their similarities, programming languages and natural languages differ in three important ways: size, rigidity, and literalness.

Size

Programming languages have small vocabularies and simple syntax and semantics compared to natural languages. Thus, their basic elements are not hard to learn.

Rigidity

In a programming language one must get the syntax absolutely correct, whereas a grammatically incorrect English sentence is usually comprehensible. This strict requirement of correctness often makes writing programs difficult for beginners, though no more difficult than writing grammatically correct sentences in English or any other natural language.

Literalness

When we give a friend instructions in English, we can be a little vague, relying on the friend to fill in the details. In a programming language, we must be exhaustively thorough. Computers follow instructions in a very literal manner. They do exactly what they are told—no more and no less. When people blame problems on computer errors, they should more accurately blame sloppy programming. This last difference is the one that makes programming difficult even for experienced programmers.

Although programming languages are simpler than human languages, the task of writing programs is challenging. It is difficult to express complex ideas using the limited syntax and semantics of a programming language.

EXERCISE 3.1

1. What is the vocabulary of a language? Give an example of an item in the vocabulary of Java.
2. Give an example of a syntax rule in Java.
3. What does the expression `(x + y) * z` mean?
4. Describe two differences between programming languages and natural languages.

3.2 Basic Java Syntax and Semantics

Having seen several Java programs in Chapter 2, we are ready for a more formal presentation of the language's basic elements. Some points have already been touched on in Chapter 2, but others are new.

Data Types

In Chapter 1, we showed that many types of information can be represented in computer memory as patterns of 0s and 1s, and as far as we know, so can information of every type. In this book, however, we are less ambitious and restrict our attention to just a few types of data. These fall into two main categories. The first category consists of what Java calls *primitive data types* and includes numbers (both integer and floating-point), characters (such as *A*, *B*, and *C*), and Booleans (restricted to the logical values `true` and `false`). The second category consists of objects, for instance, scanners. Strings are also in this second category.

Syntax

Java's syntax for manipulating primitive data types differs distinctly from the syntax for manipulating objects. Primitive data types are combined in expressions involving operators, such as addition and multiplication. Objects, on the other hand, are sent messages. In addition,

objects must be instantiated before use, and there is no comparable requirement for primitive data types. For more information about data types, see Appendix B.

Actually, this concise picture is confused slightly by strings, which on the one hand are objects and are sent messages, but on the other hand do not need to be instantiated and can be combined using something called the *concatenation operator*.

Numbers

We close this subsection with a few details concerning the primitive data types for representing numbers, or *numeric data types*, as they are called for short. Java includes six numeric data types, but we restrict ourselves to just two. These are `int` (for integer) and `double` (for floating-point numbers—numbers with decimals). The range of values available with these two data types is shown in Table 3-2.

TABLE 3-2
Some Java numeric data types

TYPE	STORAGE REQUIREMENTS	RANGE
int	4 bytes	–2,147,483,648 to 2,147,483,647
double	8 bytes	–1.79769313486231570E+308 to 1.79769313486231570E+308

The other numeric data types are *short* (2 bytes for small integers), *long* (4 bytes for large integers), *byte*, and *float* (4 bytes for smaller, less precise floating-point numbers). The data types for Booleans and characters will be discussed in Chapters 6 and 7, respectively. A complete table of the storage requirements and range of each type appears in Appendix B.

Numeric calculations are a central part of most, though not all, programs, so we will become very familiar with the numeric data types. Programs that manipulate numeric data types often share a common format: input numeric data, perform calculations, output numeric results. The temperature conversion program in Chapter 2 adhered to this format.

EXERCISE 3.2

1. What is the difference between `double` and `int` data types?

2. How does the syntax for manipulating numeric data types and objects differ?

Literals

Literals are items in a program whose values do not change. They are restricted to the primitive data types and strings. Examples from the temperature conversion program in Chapter 2 included the numbers 5.0 and 9.0 and the string "Enter degrees Fahrenheit: ". Table 3-3 gives other examples of numeric literals (note that numeric literals never contain commas).

TABLE 3-3
Examples of numeric literals

EXAMPLE	DATA TYPE
51	an integer
–31444843	a negative integer
3.14	a floating-point number (double)
5.301E5	a floating-point number equivalent to $5.301 * 10^5$, or 530,100
5.301E–5	a floating-point number equivalent to $5.301 * 10^{-5}$, or 0.00005301 (double)

The last two examples in Table 3-3 are written in what is called *exponential* or *scientific notation* and are expressed as a decimal number followed by a power of 10. The letter *E* can be written in upper- or lowercase.

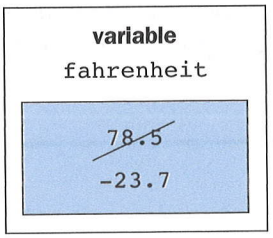
EXERCISE 3.2 Continued

3. Convert the following floating-point numbers to exponential notation:
   a. 23.5
   b. 0.046
4. Convert the following numbers from exponential notation to floating-point notation:
   a. 32.21E4
   b. 55.6E–3
5. Give two examples of string literals.

Variables and Their Declarations

A variable is an item whose value can change during the execution of a program. A variable can be thought of as a named location or cell in the computer's memory. Changing the value of a variable is equivalent to replacing the value that was in the cell with another value (Figure 3-1). For instance, at one point in a program we might set the value of the variable `fahrenheit` to 78.5. Later in the program, we could set the variable to another value such as –23.7. When we do this, the new value replaces the old one.

FIGURE 3-1
Changing the value of a variable



Although the value of a variable can change, the type of data it contains cannot; in other words, during the course of a program a specific variable can hold only one type of data. For instance, if it initially holds an integer, then it can never hold a floating-point number, and if it initially holds a reference to a pen, it can never hold a reference to a scanner.

Declarations

Before using a variable for the first time, the program must declare its type. This is done in a *variable declaration statement*, as illustrated in the following code:

```
int age;
double celsius;
String name;
Scanner reader;
```

The type appears on the left and the variable's name on the right. Frequently, we will speak of a *variable's type*, meaning the type indicator that appears on the left. Thus, we will say that `celsius` is a `double`.

It is permitted to declare several variables in a single declaration and simultaneously to assign them initial values. For instance, the following code segment initializes the variables `z`, `q`, `pi`, `name`, and `reader`:

```
int x, y, z = 7;
double p, q = 1.41, pi = 3.14, t;
String name = "Bill Jones";
Scanner reader = new Scanner(System.in);
```

The last statement declares the object variable `reader`, instantiates or creates a `Scanner` object that is attached to the keyboard input stream, `System.in`, and finally assigns the object to the variable. Instantiation takes the form

```
new <name of class>(<zero or more parameters>)
```

Object instantiation will be explained in detail in Chapter 5.

Constants

When initializing a variable, we occasionally want to specify that its value cannot change thereafter. This seems somewhat contradictory but is useful, as we shall see later. The next example illustrates how to do this:

```
final double SALES_TAX_RATE = .0785;
```

The keyword here is `final`, and a variable declared in this way is called a *constant*. It is customary to write the names of constants in uppercase. Any attempt to change the value of a constant after it is initialized is flagged by the compiler as an error.

EXERCISE 3.2 Continued

6. Why is a variable called a variable?

7. Return to the programs in Chapter 2 and find an example of each of the different types of variables. Which of the types listed in this subsection are not included?

EXERCISE 3.2 Continued

8. Declare a floating-point variable called `payRate` and simultaneously initialize it to $35.67.
9. Declare three integer variables (`a`, `b`, `c`) in a single declaration and simultaneously initialize `b` to `4`.
10. Give two examples of data that cannot be stored in a variable of type `int`.
11. There are approximately 2.2 pounds in a kilogram. Name and declare a constant to represent this value.

Assignment Statements

An assignment statement has the following form:

```
<variable> = <expression>;
```

where the value of the expression on the right is assigned to the variable on the left. For instance,

```
double celsius, fahrenheit;
String name;
Scanner reader;
. . .
fahrenheit = reader.nextDouble();
celsius = (fahrenheit - 32.0) * 5.0 / 9.0;
name = "Bill Smith";
reader = new Scanner(System.in);
```

Arithmetic Expressions

An ***arithmetic expression*** consists of operands and operators combined in a manner familiar from algebra. The usual rules apply:

- Multiplication and division are evaluated before addition and subtraction; that is, multiplication and division have higher precedence than addition and subtraction.
- Operators of equal precedence are evaluated from left to right.
- Parentheses can be used to change the order of evaluation.

Unlike in algebra, multiplication must be indicated explicitly: thus, `a * b` cannot be written as `ab`. Binary operators are placed between their operands (`a * b`, for example), whereas unary operators are placed before their operands (`-a`, for example). Table 3-4 shows several operands from the conversion program in Chapter 2, and Table 3-5 shows some common operators and their precedence.

TABLE 3-4
Examples of operands

TYPE	EXAMPLE
Literals	`32.0  5.0  9.0`
Variables	`fahrenheit  celsius`
Parenthesized expressions	`(fahrenheit - 32.0)`

TABLE 3-5
Common operators and their precedence

OPERATOR	SYMBOL	PRECEDENCE (FROM HIGHEST TO LOWEST)	ASSOCIATION
Grouping	( )	1	Not applicable
Method selector	.	2	Left to right
Unary plus	+	3	Not applicable
Unary minus	-	3	Not applicable
Instantiation	new	3	Right to left
Cast	(double) (int)	3	Right to left
Multiplication	*	4	Left to right
Division	/	4	Left to right
Remainder or modulus	%	4	Left to right
Addition	+	5	Left to right
Subtraction	-	5	Left to right
Assignment	=	10	Right to left

Division

Several points concerning operators need explanation. The semantics of division are different for integer and floating-point operands. Thus,

```
5.0 / 2.0    yields    2.5
5 / 2 yields 2 (a quotient in which the fractional portion of the answer is
  simply dropped)
```

Modulus

The operator `%` yields the remainder obtained when one number is divided by another. Thus,

```
9 % 5 yields 4
9.3 % 5.1 yields 4.2
```

Precedence

When evaluating an expression, Java applies operators of higher precedence before those of lower precedence, unless overridden by parentheses. The highest precedence is 1.

```
3 + 5 * 3    yields 18
-3 + 5 * 3   yields 12
```

```
+3 + 5 * 3   yields 18 (use of unary + is uncommon)
3 + 5 * -3   yields -12
3 + 5 * +3   yields 18 (use of unary + is uncommon)
(3 + 5) * 3  yields 24
3 + 5 % 3    yields 5
(3 + 5) % 3  yields 2
```

Association

The column labeled "Association" in Table 3-5 indicates the order in which to perform operations of equal precedence. Thus,

```
18 - 3 - 4   yields 11
18 / 3 * 4   yields 24
18 % 3 * 4   yields 0
a = b = 7;   assigns 7 to b and then b to a
```

More Examples

Some more examples of expressions and their values are shown in Table 3-6. In this table, we see the application of two fairly obvious rules governing the use of parentheses:

1. Parentheses must occur in matching pairs.
2. Parenthetical expressions may be nested but must not overlap.

TABLE 3-6
Examples of expressions and their values

EXPRESSION	SAME AS	VALUE
3 + 4 – 5	7 – 5	2
3 + (4 – 5)	3 + (–1)	2
3 + 4 * 5	3 + 20	23
(3 + 4) * 5	7 * 5	35
8 / 2 + 6	4 + 6	10
8 / (2 + 6)	8 / 8	1
10 – 3 – 4 – 1	7 – 4 – 1	2
10 – (3 – 4 – 1)	10 – (–2)	12
(15 + 9) / (3 + 1)	24 / 4	6
15 + 9 / 3 + 1	15 + 3 + 1	19
(15 + 9) / ((3 + 1) * 2)	24 / (4 * 2) 24 / 8	3
(15 + 9) / (3 + 1) * 2	24 / 4 * 2 6 * 2	12

EXERCISE 3.2 Continued

12. Assume that the integer variable `x` is 5 and the integer variable `y` is 10. Give the values of the following expressions:

a. x + y * 2

b. x – y + 2

c. (x + y) * 2

d. y % x

13. Find the syntax errors in the following expressions:

a. a – * b + c

b. – (a + b) * c)

c. ()

Maximum, Minimum, and Arithmetic Overflow

Numeric data types in any programming language support a finite range of values. For example, values of type `int` in Java range from a minimum of –2,147,483,648 to a maximum of 2,147,483,647. Instead of having to remember these numbers, the programmer can refer to them with the constants `Integer.MIN_VALUE` and `Integer.MAX_VALUE`, respectively. The same constants are included in the `Double` class for the bounds of double values.

It is natural to ask what would happen if a program tried to add 1 to the maximum integer value. This would result in a condition known as arithmetic overflow. Subtracting 1 from the minimum value would produce the same error. Programs written in some languages would halt with a run-time error, whereas others would continue after ruining part of the computer's operating system. The JVM simply inverts the sign of the number and allows the program to continue. Thus, adding 1 to `Integer.MAX_VALUE` would produce `Integer.MIN_VALUE`. So, if you see large negative integers in your program's output, you might have this type of error.

Mixed-Mode Arithmetic

When working with a handheld calculator, we do not give much thought to the fact that we intermix integers and floating-point numbers. This is called *mixed-mode arithmetic*. For instance, if a circle has radius 3, we compute the area as follows:

```
3.14 * 3 * 3
```

In Java, when there is a binary operation on operands of different numeric types, the less inclusive type (`int`) is temporarily and automatically converted to the more inclusive type (`double`) before the operation is performed. Thus, in

```
double d;
d = 5.0 / 2;
```

the value of `d` is computed as `5.0/2.0`, yielding `2.5`. However, problems can arise when using mixed-mode arithmetic. For instance

```
3 / 2 * 5.0  yields   1 * 5.0   yields   5.0
```

whereas

```
3 / 2.0 * 5  yields   1.5 * 5   yields   7.5
```

Mixed-mode assignments are also allowed, provided the variable on the left is of a more inclusive type than the expression on the right. Otherwise, a syntax error occurs, as shown in the following code segment:

```
double d;
int i;
i = 45; ← OK, because we assign an int to an int.
d = i;  ← OK, because d is more inclusive than i. The value 45.0 is stored in d.
i = d;  ← Syntax error because i is less inclusive than d.
```

EXERCISE 3.2 Continued

14. Assume that `x` is 4.5 and `y` is 2. Write the values of the following expressions:

a. x / y

b. y / x

c. x % y

15. Assume that `x` and `y` are of type `double` and `z` is of type `int`. For each of the following assignment statements, state which are valid and which produce syntax errors:

a. x = z

b. x = y * z

c. z = x + y

Casting to `int` and `double`

The difficulties associated with mixed-mode arithmetic can be circumvented using a technique called *casting*, which allows one data type to be explicitly converted to another. For instance, consider the following example:

```
int i;
double d;

i = (int)3.14;          ← i equals 3, truncation toward 0
d = (double)5 / 4;      ← d equals 1.25
```

The cast operator, either `(int)` or `(double)`, appears immediately before the expression it is supposed to convert. The `(int)` cast simply throws away the digits after the decimal point, which has the effect of truncation toward 0.

Precedence

The cast operator has high precedence (see Table 3-5) and must be used with care, as illustrated in the following code:

```
double x, y;

x = (double)5 / 4;      ← x equals 5.0 / 4 equals 1.25
y = (double)(5 / 4);    ← y equals (double)(1) equals 1.0
```

Rounding

The cast operator is useful for rounding floating-point numbers to the nearest integer:

```
int m, n;
double x, y;

x = . . . ;                 ← some positive value is assigned to x
m = (int)(x + 0.5);

y = - . . .;                ← some negative value is assigned to y
n = (int)(x - 0.5);
```

Other numeric casts such as `(char)` and `(float)` are discussed in Appendix B.

EXERCISE 3.2 Continued

16. Assume that `x` is of type `double` and `y` is of type `int`. Also assume that `x` is 4.5 and `y` is 2. Write the values of the following expressions:

a. `(int) x * y`

b. `(int) (x * y)`

17. Assume that `x` is of type `double` and `y` is of type `int`. Write a statement that assigns the value contained in `x` to `y` after rounding this value to the nearest whole number.

String Expressions and Methods

Strings are used in programs in a variety of ways. As already seen, they can be used as literals or assigned to variables. Now we see that they can be combined in expressions using the concatenation operator, and they also can be sent messages.

Simple Concatenation

The concatenation operator uses the plus symbol (+). The following is an example:

```
String firstName,                                    // declare four string
     lastName,                                       // variables
     fullName,
     lastThenFirst;

firstName = "Bill";                                  // initialize firstName
lastName = "Smith";                                  // initialize lastName

fullName = firstName + " " + lastName;               // yields "Bill Smith"
lastThenFirst = lastName + ", " + firstName          // yields "Smith, Bill"
```

Concatenating Strings and Numbers

Strings also can be concatenated to numbers. When this occurs, the number is automatically converted to a string before the concatenation operator is applied:

```
String message;
int x = 20, y = 35;
```

```
message = "Bill sold " + x + " and Sylvia sold " + y + " subscriptions.";
// yields "Bill sold 20 and Sylvia sold 35 subscriptions."
```

Precedence of Concatenation

The concatenation operator has the same precedence as addition, which can lead to unexpected results:

```
"number " + 3 + 4      → "number 3" + 4    → "number 34"
"number " + (3 + 4)    → "number " + 7     → "number 7"
"number " + 3 * 4      → "number " + 12    → "number 12"
3 + 4 + "number"       → 7 + "number"      → "7 number"
```

Escape Character

String literals are delimited by quotation marks ("..."), which presents a dilemma when quotation marks are supposed to appear inside a string. Placing a special character before the quotation mark, indicating the quotation mark is to be taken literally and not as a delimiter, solves the problem. This special character, also called the *escape character*, is a back slash (\).

```
message = "As the train left the station, " +
          "the conductor yelled, \"All aboard.\"";
```

Other Uses for the Escape Character

The escape character also is used when including other special characters in string literals. The sequence backslash-t (\t) indicates a tab character, and backslash-n (\n) indicates a newline character. These special sequences involving the backslash character are called ***escape sequences***. The following code gives an example of the use of escape sequences followed by the output generated by the code:

Code

```
System.out.print ("The room was full of animals: \n" +
                  "\tdogs,\n\tcats, and\n\tchimpanzees.\n";
```

Output

```
The room was full of animals:
       dogs,
       cats, and
       chimpanzees.
```

Escaping the Escape Character

In solving one problem, we have introduced another. The backslash is the designated escape character, but sometimes a string must contain a backslash. This is accomplished by placing two backslashes in sequence:

```
path = "C:\\Java\\Ch3.doc";       • yields the string C:\Java\Ch3.doc
```

There are several other escape sequences, but we omit them from this discussion (see Appendix B for further details).

The `length` Method

Strings are objects and implement several methods. In this chapter, we consider only the `length` method and defer discussions of others until Chapters 6 and 11 (for users of the Introductory and Comprehensive texts). A string returns its length in response to a `length` message:

```
String theString;
int theLength;

theString = "The cat sat on the mat.";
theLength = theString.length();          ← yields 23
```

EXERCISE 3.2 Continued

18. Assume that x refers to the string "`Wizard`" and y refers to the string "`Java`". Write the values of the following expressions:

a. `y + x`

b. `y + y.length() + x`

c. `y + "\n" + x + "\n"`

19. Declare a variable of type `String` called `myInfo` and initialize it to your name, address, and telephone number. Each item of information in this string should be followed by a newline character.

Methods, Messages, and Signatures

Classes implement methods, and objects are instances of classes. An object can respond to a message only if its class implements a corresponding method. To correspond, the method must have the same name as the message. Thus a pen object responds to the `move` message because the `StandardPen` class defines a `move` method.

Messages are sometimes accompanied by parameters and sometimes not:

```
double x = reader .nextDouble();    // No parameters expected
System.out.println(50.5);           // One parameter expected
```

The parameters included when a message is sent must match exactly in number and type the parameters expected by the method. For instance, the `Math.sqrt` method expects a single parameter of type `double`.

```
double d = 24.6;

Math.sqrt(d)           // Perfect! A parameter of type double is expected.
Math.sqrt(2.0 * d)     // Perfect! The expression yields a double.
Math.sqrt(4)           // Fine! Integers can stand in for doubles.
Math.sqrt();           // Error! A parameter is needed.
Math.sqrt(6.7, 3.4);   // Error! One parameter only please.
Math.sqrt("far");      // Error! A string parameter is NOT acceptable.
```

Some methods return a value and others do not. The `println` method does not return a value; however, the method `nextDouble` in class `Scanner` does:

```
Scanner reader = new Scanner();
double x;

x = reader.nextDouble();          // Returns the number entered by the user.
```

To use a method successfully we must know

- What type of value it returns
- Its name
- The number and type of the parameters it expects

A method's name and the types and number of its parameters are called the method's *signature*. From now on, when we introduce a new class, we will make a point of listing method signatures together with brief descriptions of what the methods do. Following are two examples, the first from the `Scanner` class and the second from the `PrintStream` class:

```
double nextDouble()          Returns a double entered by the user at the
                             keyboard.
void   println (double n)    Writes n to the print stream.
```

The word `void` indicates that the method does not return a value.

EXERCISE 3.2 Continued

20. What is the difference between a message and a method?

21. Describe the purpose of each item of information that appears in a method's signature.

User-Defined Symbols

Variable and program names are examples of user-defined symbols. You will see other examples later in the book. We now explain the rules for forming or naming user-defined symbols. These names must consist of a letter followed by a sequence of letters and/or digits. Letters are defined to be

- A ... Z
- a ... z
- _ and $
- Symbols that denote letters in several languages other than English

Digits are the characters `0 ... 9`. Names are case-sensitive; thus, `celsius` and `Celsius` are different names.

Some words cannot be employed as user-defined symbols. These words are called *keywords* or ***reserved words*** because they have special meaning in Java. Table 3-7 shows a list of Java's reserved words. You will encounter most of them by the end of the book. They appear in blue in program code examples. These words are case-sensitive also, thus "import" is a reserved word but "Import" and "IMPORT" are not.

TABLE 3-7
Java's reserved words

abstract	double	int	static
assert	false	strictfp	true
boolean	else	interface	super
break	extends	long	switch
byte	final	native	synchronized
case	finally	new	this
catch	float	null	throw
char	for	package	throws
class	goto	private	transient
const	if	protected	try
continue	implements	public	void
default	import	return	volatile
do	instanceof	short	while

Here are examples of valid and invalid variable names:

Valid Names `surfaceArea3` `_$_$$$`

Invalid Names `3rdPayment` `pay.rate` `abstract`

The first invalid name begins with a digit. The second invalid name contains a period. The third invalid name is a reserved word.

Well-chosen variable names greatly increase a program's readability and maintainability; consequently, it is considered good programming practice to use meaningful names such as

`radius` rather than `r`

`taxableIncome` rather than `ti`

When forming a compound variable name, programmers usually capitalize the first letter of each word except the first. For instance,

`taxableIncome` rather than

`taxableincome` or

`TAXABLEINCOME` or

`TaxableIncome`

On the other hand, all the words in a program's name typically begin with a capital letter, for instance, `ComputeEmployeePayroll`. Finally, constant names usually are all uppercase. The goal of these rules and of all stylistic conventions is to produce programs that are easier to understand and maintain.

EXERCISE 3.2 Continued

22. State whether each of the following are valid or invalid user-defined symbols in Java:

a. pricePerSquareInch

b. student2

c. 2GuysFromLexington

d. PI

e. allDone?

23. Write names for the following items that follow good programming practice:

a. A variable that represents the diameter of a circle

b. A constant that represents the standard deduction for an income tax return

c. A method that draws a rectangle

Packages and the `import` statement

Programmers seldom write programs from scratch. Instead, they rely heavily on code written by many other programmers, most of whom they will never meet. Fortunately, Java provides a mechanism called a ***package*** that makes it easy for programmers to share code. Having written a class or group of classes that provide some generally useful service, a programmer can collect the classes together in a package. Other programmers who want to use the service can then import classes from the package. The programs in Chapter 2 illustrated the use of a package called `java.util`. The Java programming environment typically includes a large number of standard packages, some of which we use in subsequent chapters. These packages are standard because we expect to find them in every Java programming environment. Some packages are nonstandard and are created by their authors to support various applications.

When using a package, a programmer imports the desired class or classes. The general form of an `import` statement is

```
import x.y.z;
```

where

- `x` is the overall name of the package.
- `y` is the name of a subsection within the package.
- `z` is the name of a particular class in the subsection.

It is possible to import all the classes within a subsection at once; however, we do not usually do so. The statement to import all the classes with a subsection looks like this:

```
import x.y.*;
```

In general, a package can have any number of subsections, including zero, which is the case for several packages used in this book, and each subsection can in turn have any number of subsubsections, and so on. When used, an asterisk (*) can appear only at the lowest level. The asterisk is used to make available all of the classes in a package.

EXERCISE 3.2 Continued

24. Describe the role of the items `x`, `y`, and `z` in the statement `import x.y.z;`.

25. What happens when the computer executes the statement `import x.y.*;`?

3.3 Terminal I/O for Different Data Types

Objects support terminal input and output (I/O). An instance of the class `Scanner` supports input and the object `System.out` supports output. The latter object is an instance of the class `PrintStream`. This class, together with a number of others, is available to Java programmers without specifying their names in import statements. Although `System.out` is an instance of the class `PrintStream`, do not try to instantiate this class until you become familiar with working in Java files.

Table 3-8 summarizes some of the methods in the class `Scanner`. The object `System.out` understands two messages, `print` and `println`. Both messages expect a single parameter, which can be of any type, including an object; however, we postpone using an object as a parameter until Chapter 5.

TABLE 3-8
Methods in class `Scanner`

METHOD	DESCRIPTION
`double nextDouble()`	Returns the first `double` in the input line. Leading and trailing spaces are ignored.
`int nextInt()`	Returns the first integer in the input line. Leading and trailing spaces are ignored.
`String nextLine()`	Returns the input line, including leading and trailing spaces. *Warning*: A leading newline is returned as an empty string.

The following program illustrates the major features of terminal I/O:

```
// Example 3.1: tests 3 types of input data

import java.util.Scanner;

public class TestTerminalIO {

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

      String name;
      int age;
      double weight;
```

```
        System.out.print ("Enter your name (a string): ");
        name = reader.nextLine();

        System.out.print ("Enter your age (an integer): ");
        age = reader.nextInt();

        System.out.print ("Enter your weight (a double): ");
        weight = reader.nextDouble();

        System.out.println ("Greetings " + name +
                            ". You are " + age +
                            " years old and you weigh " + weight +
                            " pounds.");

    }
}
```

When the program encounters an input statement—for instance, `reader.nextInt();`—it pauses and waits for the user to press Enter, at which point the `reader` object processes the user's input. The interaction with the user looks something like this, where the user's input is shown in boldface and the use of the Enter key is shown in italics:

```
Enter your name (a string): Carole JonesEnter
Enter your age (an integer): 45Enter
Enter your weight (a double): 130.6Enter
Greetings Carole Jones. You are 45 years old and you weigh 130.6 pounds.
```

The example program in this section reads a line of text followed by two numbers. Let's consider a code segment that reads numbers followed by a line of text, as follows:

```
System.out.print("Enter your age (an integer): ");
age = reader.nextInt();

System.out.print("Enter your weight (a double): ");
weight = reader.nextDouble();

System.out.print("Enter your name (a string): ");
name = reader.nextLine();
```

The program will receive the numbers as entered, but unfortunately the string input on the last line of code will be empty (""). The reason for this is that the method `nextDouble`, which input the last number two steps earlier, ignored but did not consume the newline that the user entered following the number. Therefore, this newline character was waiting to be consumed by the next call of `nextLine`, which was expecting more data! To avoid this problem, you should either input all the lines of text before the numbers or, when that is not feasible, run an extra call of `nextLine` after numeric input to eliminate the trailing newline character. Here is code that uses the second alternative:

```
System.out.print("Enter your age (an integer): ");
age = reader.nextInt();

System.out.print("Enter your weight (a double): ");
weight = reader.nextDouble();
```

```
reader.nextLine();    // Consume the newline character

System.out.print("Enter your name (a string): ");
name = reader.nextLine();
```

EXERCISE 3.3

1. Write code segments that perform the following tasks:
   a. Prompt the user for an hourly wage and read the wage into a `double` variable `wage`.
   b. Prompt the user for a Social Security number and read this value into the `String` variable `ssn`.
2. What is the purpose of the method `nextInt()`?
3. Explain what happens when a program reads a number from the keyboard and then attempts to read a line of text from the keyboard.

Programming Skills

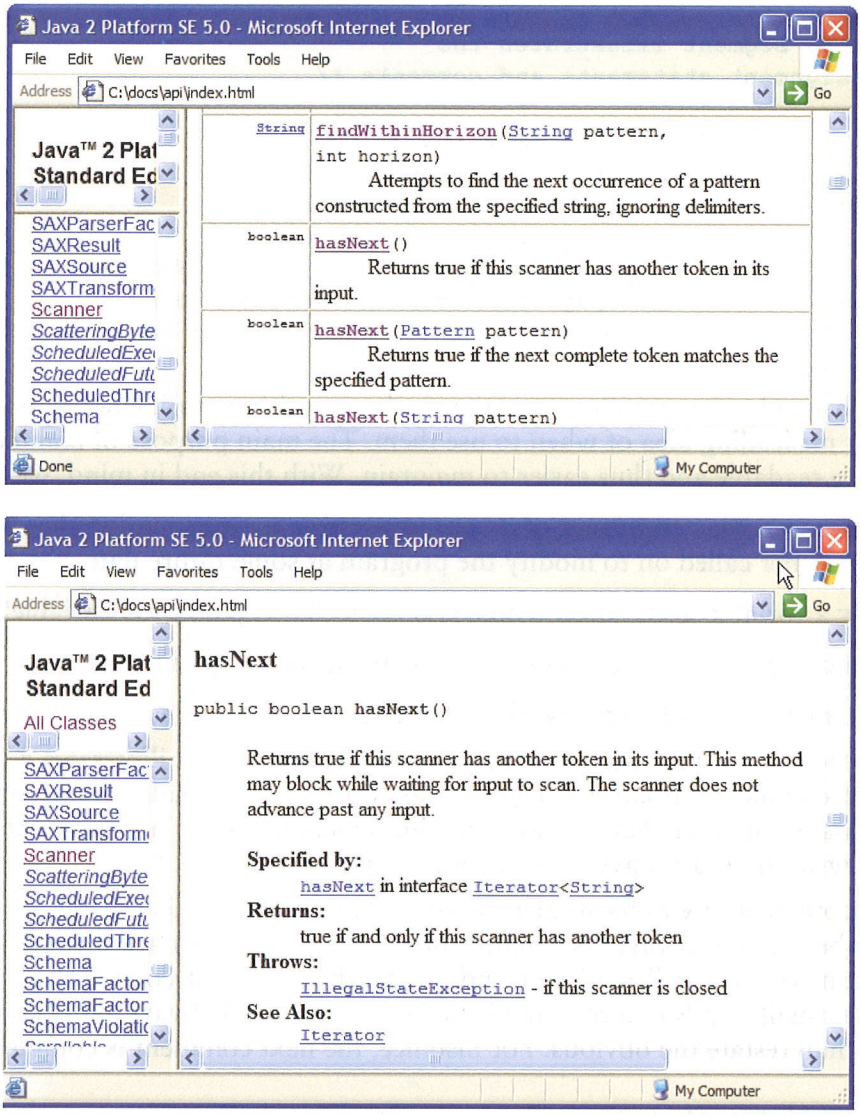
THE JAVA API AND `javadoc`

Most programming languages are defined in language reference manuals. These manuals show the syntax, the semantics, and how to use the basic features of the language. For Java, this documentation can be found in the Java Application Programming Interface (API). You can browse the Java API online at Sun's Web site or download the API for your particular version of Java to browse locally on your own computer.

You can browse for a given class's information by selecting the name of the class from a master list of classes or by selecting its package from a master list of packages. You can quickly drill down from there to a list of the class's methods and finally to the information for an individual method. The screen shots in Figure 3-2 show part of the method list in the `Scanner` class and the detailed information for the method `hasNext()`, respectively. The API for Java is enormous, with dozens of packages, hundreds of classes, and thousands of methods. However, the browser allows you to locate most information on a given item with a few mouse clicks, and links provide helpful cross-references to related items.

In addition to the `javac` and `java` commands for compiling and running programs, Sun provides a `javadoc` command that allows you to create browsable documentation for your own code. We explore how to create this documentation when we examine interfaces for user-defined classes in Chapter 10.

FIGURE 3-2
A portion of Sun's documentation for the Java API



3.4 Comments

When we first write a program, we are completely familiar with all its nuances; however, six months later, when we or someone else has to modify it, the code that was once so clear often seems confusing and mysterious. There is, however, a technique for dealing with this situation. The remedy is to include comments in the code. ***Comments*** are explanatory sentences inserted in a program in such a manner that the compiler ignores them. There are two styles for indicating comments:

- End of line comments: these include all of the text following a double slash (//) on any given line; in other words, this style is best for just one line of comment.
- Multiline comments: these include all of the text between an opening /* and a closing */.

Comments appear in green in this book. The following code segment illustrates the use of both kinds of comments:

```
/* This code segment illustrates the
use of assignment statements and comments */

a = 3;              // assign 3 to variable a
b = 4;              // assign 4 to variable b
c = a + b;          // add the number in variable a
                    //    to the number in variable b
                    //    and assign the result, 7, to variable c
c = c * 3;          // multiply the number in variable c by 3
                    //    and assign the result, 21, to variable c
```

Although this code segment illustrates the mechanics of how to include comments in a program, it gives a misleading idea of when to use them. The main purpose of comments is to make a program more readable and thus easier to maintain. With this end in mind, we usually

- Begin a program with a statement of its purpose and other information that would help orient a programmer called on to modify the program at some future date.
- Accompany a variable declaration with a comment that explains the variable's purpose.
- Precede major segments of code with brief comments that explain their purpose.
- Include comments to explain the workings of complex or tricky sections of code.

The case study in the next section follows these guidelines and illustrates a reasonable and helpful level of comments. Because the programs in this book usually are accompanied by an extensive discussion of what they do, we sometimes include few or no comments; however, the programs you write should always be well commented.

Too many comments are as harmful as too few because, over time, the burden of maintaining the comments becomes excessive. No matter how many comments are included in a program, future programmers must still read and understand the region of code they intend to modify. Common sense usually leads to a reasonable balance. We should always avoid comments that do nothing more than restate the obvious. For instance, the next comment is completely pointless:

```
a = 3;      // assign 3 to variable a. Duh!
```

The best-written programs are self-documenting; that is, the reader can understand the code from the symbols used and from the structure and overall organization of the program.

EXERCISE 3.4

1. Describe the difference between an end-of-line comment and a multiline comment.
2. State two rules of thumb for writing appropriate comments in a program.

Case Study 1: Income Tax Calculator

It is now time to write a program that illustrates some of the concepts we have been presenting. We do this in the context of a case study that adheres to the software development life cycle discussed in Chapter 1. This life cycle approach may seem overly elaborate for small programs, but it scales up well when programs become larger.

Each year nearly everyone with an income faces the unpleasant task of computing his or her income tax return. If only it could be done as easily as suggested in this case study.

Request

Write a program that computes a person's income tax.

Analysis

Here is the relevant tax law (mythical in nature):

- There is a flat tax rate of 20 percent.
- There is a $10,000 standard deduction.
- There is a $2000 additional deduction for each dependent.
- Gross income must be entered to the nearest penny.
- The income tax is expressed as a decimal number.

The user inputs are the gross income and number of dependents. The program calculates the income tax based on the inputs and the tax law and then displays the income tax. Figure 3-3 shows the proposed terminal user interface. Characters in boldface indicate user inputs. The program prints the rest. The inclusion of a user interface at this point is a good idea because it allows the customer and the programmer to discuss the intended program's behavior in a context understandable to both.

FIGURE 3-3
The user Interface for the income tax calculator

```
Enter the gross income: 50000.50
Enter the number of dependents: 4
The income tax is $6400.1
```

Design

During analysis, we specify what a program is going to do, and during design we describe how it is going to do it. This involves writing the algorithm used by the program. *Webster's New Collegiate Dictionary* defines an *algorithm* as "a step-by-step procedure for solving a problem or accomplishing some end." A recipe in a cookbook is a good example of an algorithm. Program algorithms are often written in a somewhat stylized version of English called ***pseudocode***. The following is the pseudocode for our income tax program:

```
read grossIncome
read numDependents
compute taxableIncome = grossIncome - 10000 - 2000 * numDependents
compute incomeTax = taxableIncome * 0.20
print incomeTax
```

Although there are no precise rules governing the syntax of pseudocode, you should strive to describe the essential elements of the program in a clear and concise manner. Over time, you will develop a style that suits you.

Implementation

Given the preceding pseudocode, an experienced programmer now would find it easy to write the corresponding Java program. For a beginner, on the other hand, writing the code is the most difficult part of the process. The following is the program:

```
/*Case study 3.1: an income tax calculator
Compute a person's income tax
1. Significant constants
       tax rate
       standard deduction
       deduction per dependent
2. The inputs are
       gross income
       number of dependents
3. Computations:
       net income = gross income - the standard deduction -
                    a deduction for each dependent
       income tax = is a fixed percentage of the net income
4. The outputs are
       the income tax
*/

import java.util.Scanner;

public class IncomeTaxCalculator{

   public static void main(String [] args){

      // Constants
      final double TAX_RATE = 0.20;
      final double STANDARD_DEDUCTION = 10000.0;
      final double DEPENDENT_DEDUCTION = 2000.0;

      Scanner reader = new Scanner(System.in);

      double grossIncome;          // the gross income (input)
      int  numDependents;          // the number of dependents (input)
      double taxableIncome;        // the taxable income (calculated)
      double incomeTax;            // the income tax (calculated and
                                   // output)

      // Request the inputs
      System.out.print("Enter the gross income: ");
      grossIncome = reader.nextDouble();
      System.out.print("Enter the number of dependents: ");
      numDependents = reader.nextInt();
```

```
        // Compute the income tax
        taxableIncome = grossIncome - STANDARD_DEDUCTION -
                        DEPENDENT_DEDUCTION * numDependents;
        incomeTax = taxableIncome * TAX_RATE;

        // Display the income tax
        System.out.println("The income tax is $" + incomeTax);
    }
}
```

Notice that we have used mixed-mode arithmetic, but in a manner that does not produce any undesired effects.

Computer Ethics

COMPUTER VIRUSES

A ***virus*** is a computer program that can replicate itself and move from computer to computer. Some programmers of viruses intend no harm; they just want to demonstrate their prowess by creating viruses that go undetected. Other programmers of viruses intend harm by causing system crashes, corruption of data, or hardware failures.

Viruses migrate by attaching themselves to normal programs, and then become active again when these programs are launched. Early viruses were easily detected if one had detection software. This software examined portions of each program on the suspect computer and could repair infected programs.

Viruses and virus detectors have coevolved through the years, however, and both kinds of software have become very sophisticated. Viruses now hide themselves better than they used to; virus detectors can no longer just examine pieces of data stored in memory to reveal the presence or absence of a virus. Researchers have recently developed a method of running a program that might contain a virus to see whether or not the virus becomes active. The suspect program runs in a "safe" environment that protects the computer from any potential harm. As you can imagine, this process takes time and costs money. For an overview of the history of viruses and the new detection technology, see Carey Nactenberg, "Computer Virus-Antivirus Coevolution," *Communications of the ACM*, Volume 40, No. 1 (January 1997): 46–51.

3.5 Programming Errors

According to an old saying, we learn from our mistakes, which is fortunate because most people find it almost impossible to write even simple programs without making numerous mistakes. These mistakes, or errors, are of three types: syntax errors, run-time errors, and logic errors.

The Three Types of Errors

Syntax errors, as we learned in Chapter 2, occur when we violate a syntax rule, no matter how minor. These errors are detected at compile time. For instance, if a semicolon is missing at the end of a statement or if a variable is used before it is declared, the compiler is unable to translate the program into byte code. The good news is that when the Java compiler finds a syntax error, it prints an error message, and we can make the needed correction. The bad news, as we saw previously, is that the error messages are often quite cryptic. Knowing that there is a syntax error at a particular point in a program, however, is usually a sufficient clue for finding the error.

Run-time errors occur when we ask the computer to do something that it considers illegal, such as dividing by 0. For example, suppose that the symbols `x` and `y` are variables. Then the expression `x/y` is syntactically correct, so the compiler does not complain. However, when the expression is evaluated during execution of the program, the meaning of the expression depends on the values contained in the variables. If the variable `y` has the value 0, then the expression cannot be evaluated. The good news is that the Java runtime environment will print a message telling us the nature of the error and where it was encountered. Once again, the bad news is that the error message might be hard to understand.

Logic errors (also called *design errors* or *bugs*) occur when we fail to express ourselves accurately. For instance, in everyday life, we might give someone the instruction to turn left when what we really meant to say is to turn right. In this example,

- The instruction is phrased properly, and thus the syntax is correct.
- The instruction is meaningful, and thus the semantics are valid.
- But the instruction does not do what we intended, and thus is logically incorrect.

The bad news is that programming environments do not detect logic errors automatically. The good news is that this text offers useful tips on how to prevent logic errors and how to detect them when they occur.

Now let's look at examples of each of these types of errors.

Illustration of Syntax Errors

We have already seen examples of syntax errors in Chapter 2; however, seeing a few more will be helpful. The following is a listing of the income tax calculator program with the addition of two syntax errors. See if you can spot them. The line numbers are not part of the program but are intended to facilitate the discussion that follows the listing.

```
1   import java.util.Scanner;
2
3   public class IncomeTaxCalculator{
4     public static void main(String [] args){
5
6        final double TAX_RATE = 0.20;
7        final double STANDARD_DEDUCTION = 10000.0;
```

```
8         final double DEPENDENT_DEDUCTION = 2000.0;
9
10        Scanner reader = new Scanner(System.in);
11
12        double grossIncome;
13        int numDependents;
14        double taxableIncome;
15        double incomeTax;
16
17        System.out.print("Enter the gross income: ");
18        grossIncome = reader.readDouble();
19        System.out.print("Enter the number of dependents: ");
20        numDependents = reader.nextInt();
21
22        taxableIncome = grossincome - STANDARD_DEDUCTION -
23                        DEPENDENT_DEDUCTION * numDependents;
24        incomeTax = taxableIncome * TAX_RATE
25
26        System.out.println("The income tax is $" + incomeTax);

27    }
28 }
```

Just in case you could not spot them, the errors in the code are

- In line 22, where `grossIncome` has been misspelled as `grossincome` (remember Java is case-sensitive)
- In line 24, where the semicolon is missing at the end of the line

When the program is compiled, the terminal window contains the following error message (we could show a snapshot of the window, but we think the following plain text is more readable):

```
IncomeTaxCalculator.java:25: ';' expected
^
1 error-
```

Although the compiler says that the error occurs on line 25, the semicolon is actually missing on the previous line. The corrective action is to go back into the editor, fix this error, save the file, and compile again. Then you will see a message to the effect that the symbol `grossincome` cannot be found. You may need to repeat this process a number of times until the compiler stops finding syntax errors.

Turn to the supplemental materials and read any additional instructions that apply to your development environment.

Illustration of Run-time Errors

There are a variety of run-time errors. We now present several of the most basic. We encounter others later in the book.

Division by Integer Zero

For our first run-time error, we write a small program that attempts to perform division by 0. As is well known, division by 0 is not a well-defined operation and should be avoided. Nonetheless, we must ask what happens if we accidentally write a program that tries it. The following is a trivial program that illustrates the situation:

```
// Example 3.2: attempt to divide an int by zero

public class DivideByIntegerZero{
   public static void main(String [] args){
      int i, j = 0;
      i = 3 / j;
      System.out.println("The value of i is " + i);
   }
}
```

When we attempt to run this program, execution stops prematurely, and the following error message is displayed:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivideByIntegerZero.main(DivideByIntegerZero.java:4)
```

In this circumstance, we say that the JVM has thrown an ***exception***. The message indicates the nature of the problem, "ArithmeticException: / by zero," and its location, in line 4 of method `main`.

Division by Floating-Point Zero

Interestingly, the JVM responds rather differently when the division involves a floating-point rather than an integer 0. Consider the following nearly identical program:

```
// Example 3.3: attempt to divide a double by zero

public class DivideByFloatingPointZero {
   public static void main(String [] args) {
      double i, j = 0.0;
      i = 3.0 / j;
      System.out.println ("The value of i is " + i);
      System.out.println ("10 / i equals " + 10 / i);
  }
}
```

The program now runs to completion, and the output is

```
The value of i is Infinity
10 / i equals 0.0
```

In other words, the value of the variable `i` is considered to be `Infinity`, which is to say it falls outside the range of a `double`, and if we now divide another number by `i`, we obtain 0.

Null Pointer Exception

Not all run-time errors involve arithmetic. Variables frequently represent objects. Sending a message to such a variable before the corresponding object has been instantiated causes a null

pointer exception. Fortunately, many compilers detect the possibility of this error before it arises; however, later in this book the problem occurs in situations that the compiler cannot detect. The following is an example program with the accompanying compiler error message:

The Program

```
import java.util.Scanner;

public class Test{
   public static void main(String [] args){
      Scanner reader;
      int age;
      age = reader.nextInt();
   }
}
```

The Compiler Error Message

```
C:\Test.java:7: Variable reader may not have been initialized.
      age = reader.nextInt();
            ^
1 error
```

In this code, the compiler says that the variable `reader` may not have been initialized. If that's true (and in this case, it is), the attempt to send a message to it at runtime will cause an error. The reason the variable is not initialized is that no value has been assigned to it with an assignment statement.

No Such Method Error

The following is a final, and rather puzzling, example of a run-time error. You might not notice it, even after you examine the program and the error message.

The Program

```
// Example 3.4: a puzzling run-time error

public class PuzzlingRuntimeError{
   public static void Main(String [] args){
      System.out.println ("Hello World!");
   }
}
```

The Run-time Error Message

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

Have you spotted the problem? The word `main` has been misspelled as `Main`. Remember that Java is case-sensitive, and computers are exasperatingly literal minded. They never try to guess what you meant to say, so every mistake, no matter how small, is significant.

Illustration of Logic Errors

Incorrect output is the most obvious indication that there is a logic error in a program. For instance, suppose our temperature conversion program converts 212.0 degrees Fahrenheit to 100.06 instead of 100.0 degrees Celsius. The error is small, but we notice it. And if we do not, our customers, for whom we have written the program, surely will. We caused the problem by incorrectly using 31.9 instead of 32 in the following statement:

```
celsius = (fahrenheit - 31.9) * 5.0 / 9.0;
```

Test Data

Errors of this sort are usually found by running a program with test data for which we already know the correct output. We then compare the program's output with the expected results. If there is a difference, we reexamine the program's logic to determine why the program is not behaving as expected.

But how many tests must we perform on a program before we can feel confident that it contains no more logic errors? Sometimes the fundamental nature of a program provides an answer. Perhaps your mathematical skills are sufficiently fresh to recognize that the statement

```
celsius = (fahrenheit  - 32.0) * 5.0 / 9.0;
```

is actually the equation of a line. Because two points determine a line, if the program works correctly for two temperatures, it should work correctly for all. In general, however, it is difficult to determine how many tests are enough. But we often can break down the data into categories and test one number in each, the assumption being that if the program works correctly for one number in a category, it will work correctly for all the other numbers in the same category. Careful choice of categories then becomes crucial.

Desk Checking

We can also reduce the number of logic errors in a program by rereading the code carefully after we have written it. This is called *desk checking* and is best done when the mind is fresh. It is even possible to use mathematical techniques to prove that a program or segment of a program is free of logic errors. Because programming requires exhausting and excruciating attention to detail, avoid programming for long stretches of time or when tired, a rule you will break frequently unless you manage your time well.

Usually, we never can be certain that a program is error free, and after making a reasonable but large number of tests, we release the program for distribution and wait anxiously for the complaints. If we release too soon, the number of errors will be so high that we will lose credibility and customers, but if we wait too long, the competition will beat us to the market.

EXERCISE 3.5

1. At what point in the program development process are syntax errors, run-time errors, and logic errors detected?

2. Give an example of a run-time error and explain why the computer cannot catch it earlier in the program development process.

EXERCISE 3.5 Continued

3. State the type of error (compile-time, run-time, or logic) that occurs in each of the following pieces of code:
   a. x = y / 0
   b. x + y = z
   c. area = length + width

3.6 Debugging

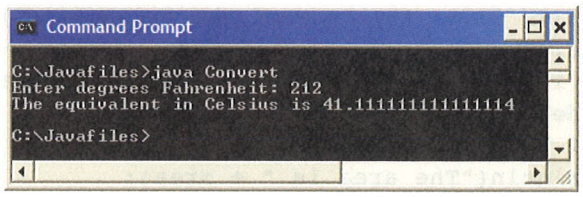
After we have established that a program contains a logic error, or *bug* as it is more affectionately called, we still have the problem of finding it. Sometimes the nature of a bug suggests its general location in a program. We then can reread this section of the program carefully with the hope of spotting the error. Unfortunately, the bug often is not located where we expect to find it, and even if it is, we probably miss it. After all, we thought we were writing the program correctly in the first place, so when we reread it, we tend to see what we were trying to say rather than what we actually said.

Programmers, as a consequence, are frequently forced to resort to a rather tedious, but powerful, technique for finding bugs. We add to the program extra lines of code that print the values of selected variables in the terminal window. Of course, we add these lines where we anticipate they will do the most good—that is, preceding and perhaps following the places in the program where we think the bug is most likely located. We then run the program again, and from the extra output, we can determine if any of the variables deviate from their expected values. If one of them does, then we know the bug is close by, but if none do, we must try again at a different point in the program. A variable's value is printed in the terminal window as follows:

```
System.out.println ("<some message>" + <variable name>);
```

Now let us try to find a bug that has been secretly inserted into the temperature conversion program. Suppose the program behaves as shown in Figure 3-4. Something is seriously wrong. The program claims that 212 degrees Fahrenheit converts to 41.1 degrees Celsius instead of the expected 100.

FIGURE 3-4
Incorrect output from the temperature conversion program



Perhaps we can find the problem by checking the value of fahrenheit just before celsius is calculated. The needed code looks like this:

```
System.out.println ("fahrenheit = " + fahrenheit);    ← This is the
                                                         debugging code
celsius = (fahrenheit - 32.0) * 5.0 / 9.0;
```

When we run the program again with the debugging code included, we get the following output:

```
Enter degrees Fahrenheit: 212
Fahrenheit = 106.0
The equivalent in Celsius is 41.111111111111114
```

We entered 212, but for some reason, the program says the value of `fahrenheit` is 106. Perhaps we should look at the surrounding code and see if we can spot the error. Here is the relevant code:

```
. . .
System.out.print ("Enter degrees Fahrenheit: ");
fahrenheit = reader.nextDouble() / 2.0;
System.out.println ("fahrenheit = " + fahrenheit);
celsius = (fahrenheit - 32.0) * 5.0 / 9.0;
. . .
```

Ah, there is the error. It looks as if the value entered by the user is divided by 2 just before it is assigned to the variable `fahrenheit`. Devious, but we cannot be deceived for long.

EXERCISE 3.6

1. Describe how one can modify code so that the cause of a logic error can be discovered.

2. The following program contains a logic error. Describe where to insert the appropriate debugging statements to help locate the error:

```
import java.util.Scanner;

public class AreaTriangle{

   public static void main(String [] args){

      double base, height, area;
      Scanner reader = new Scanner(System.in);

      System.out.print("Enter the base of the triangle: ");
      base = reader.nextDouble();
      System.out.print("Enter the height of the triangle: ");
      height = reader.nextDouble();
      area = base + height / 2;
      System.out.println("The area is " + area);

      }
}
```

Case Study 2: Count the Angels

Computers have been applied to many complex problems, from predicting the weather, to controlling nuclear power plants, to playing the best chess in the world. Now this case study

extends computing into the realm of metaphysics. Although this case study is fanciful and humorous, it illustrates important issues regarding analysis and design. During analysis and design we deliberately introduce several subtle errors, which during implementation we incorporate into the program, yet the program runs perfectly and gives no hint that there are underlying problems. As you read the case study, see if you can spot the errors. At the end we will point out what they are and make some general comments about the software development process.

Request

Write a program that determines how many angels can dance on the head of a pin.

Analysis

To solve this problem, we first consulted several prominent theologians. We learned that the pertinent factors are the size of the pinhead, the space occupied by a single angel, and the overlap between adjacent angels. Although angels are incorporeal beings, there are limits to the amount of overlap they can tolerate. Also, no region of space is ever occupied by three angels simultaneously. This seems somewhat confusing. On further questioning, the experts explained that angels have what one might call an overlap factor. If, for instance, this factor is 30 percent, then

- An angel can share at most 30 percent of its space with other angels.
- 70 percent of its space cannot be shared.
- Within any shared region, only two angels can overlap.

The inputs to the program are now fairly obvious: the radius of the pinhead, the space occupied by an angel, and the overlap factor. Based on these inputs, the program will calculate

- The area of pinhead = πr^2.
- Nonoverlapping space required by an angel = space occupied by an angel * (1 - overlap factor).
- Number of angels on pinhead = area of pinhead / nonoverlapping space required by an angel.

The proposed user interface is shown in Figure 3-5.

FIGURE 3-5
Proposed interface for the count angels program

```
Enter the radius in millimeters: 10
Enter the space occupied by an angel in square micrometers: 0.0001
Enter the overlap factor: 0.75
The number of angels = 1.256E7
```